Cache Management Policies for Semantic Caching

Themis Palpanas, Per-Åke Larson, Jonathan Goldstein

themis@cs.toronto.edu, palarson@microsoft.com, jongold@microsoft.com

Abstract

Commercial database systems make extensive use of caching to speed up query execution. Semantic caching is the idea of caching actual query results in the hope of being able to reuse them to speed up subsequent queries. This paper deals with cache management policies, which refer to policies for admission into the cache and eviction from the cache. When a query is executed, we must decide what part, if any, of the query result to add to the cache. If the cache is full, we must also decide which, if any, of the currently cached results to evict from the cache. The objective of these policies is to minimize the cost of executing the *current and future* queries or, phrased differently, maximize the benefit of the cache. The main difficulty is predicting the cost savings of future queries.

In this study we present a family of cache admission and eviction policies, named *RECYCLE*, specifically designed for semantic caching within a relational database server. Our policies were designed to be both effective (high reuse ratio) and efficient (low overhead). The policies make use of a novel type of statistics, called *access statistics*. These statistics are easy to gather and maintain, yet, they offer a solid and flexible framework for dealing with the complexity of the problem. The decision whether to cache a query result takes into account the current content of the cache and the estimated benefit of the result during its lifetime. We experimentally compare the effectiveness of our policies with several other caching policies, and show the superiority of our solution.

1. Introduction

Commercial database systems make extensive use of caching to speed up processing. All systems use one or more page caches – usually called buffer pools – that store frequently referenced database pages to reduce disk IO. On a large database server several gigabytes of memory may be used for buffer pools. Many systems also cache execution plans to avoid optimizing the same query or stored procedure multiple times. Catalog data is also frequently cached. Various caches are used during query optimization and query processing but the details vary from system to system. All these software caches are on top of the hardware caches employed by all modern processors to reduce main memory and hard disk accesses.

Semantic caching is the idea of caching actual query results in the hope of being able to reuse them to speed up subsequent queries. It is not necessary to cache only final query results – we may decide to cache the result of any subexpression of a query plan. Semantic caching can be applied in many contexts but our focus is on semantic caching within a database server. Semantic caching is an old idea but, to the best of our knowledge, it is not implemented in any commercial database product. The mechanisms and policies required for semantic caching are somewhat complex, making actual implementation a challenging task. The focus of this paper is on policies so we assume that the necessary mechanisms for saving, reading and purging cached results are in place.

Cached query results are nothing more than temporary materialized views managed entirely by the system, so much of the work on materialized views can be applied directly to semantic caching, in particular view matching and view maintenance algorithms. Maintenance considerations are different though, because we always have the option of simply discarding a cached result. To emphasize the equivalence with materialized views, we will refer to cached results as *Materialized Query Results (MQRs)*. In the same way as a materialized view, an *MQR* consists of two parts: the expression defining the result and the actual stored result.

The purpose of a semantic cache is to speed up query execution as much as possible. The speedup depends on the reuse ratio of the cache. The reuse ratio depends on several factors, some of which are outside our control. These factors are:

- **Query workload characteristics.** Caching exploits hotspots, which in the context of semantic caching means both data hotspots and query hotspots. In other words, semantic caching can only be effective if the workload contains similar queries over the same data. Furthermore, the hotspots must be reasonably stable over time. The queries must also be somewhat expensive – there is no point in caching a result that can be

computed very cheaply from base tables. These are inherent characteristics of the workload. All we can do is try to reduce the overhead of the caching mechanism when there is little to be gained by caching.

- **Update characteristics.** The more volatile the data is, the less we can expect to gain from caching because high volatility reduces the lifetime of cached data. This is true even if *MQRs* are kept up to date in the same way as materialized views or indexes because volatility increases the maintenance overhead. There is one exception to this rule, namely, when update hotspots hit a different part of the database than query hotspots. This is another inherent characteristic of the workload. All we can do is try to avoid caching frequently updated data.
- **Cache management policies.** This refers to policies for admission into the cache and eviction from the cache. When a query is executed, we must decide what part, if any, of the query result to add to the cache. If the cache is full, we must also decide which, if any, of the currently cached *MQRs* to evict from the cache.
- **Cache invalidation policies.** The contents of an *MQR* are defined by a query expression and computed from some collection of rows in the underlying tables. If one of the underlying tables is updated, the cached result may become invalid. The simplest policy is to invalidate an *MQR* whenever there are any updates to one of its underlying tables. This is a bit heavy handed though because the change may be in rows that did not and still don't contribute to the *MQR*. One can envision polices at different levels of sophistication for deciding whether an update to a base table invalidates an *MQR*.
- **Reuse algorithms.** This refers to the algorithms used for detecting whether part or all of a query can be computed from one or more *MQRs*. Again, many algorithms at different levels of speed and effectiveness are possible. Existing view matching algorithms can be applied unchanged, provided they are fast and scalable enough.

This paper deals with cache management policies. The objective of these policies is to minimize the cost of executing the *current and future* queries or, phrased differently, maximize the benefit of the cache. The main difficulty is predicting the cost savings of future queries. Suppose that we actually knew the sequence of future queries and updates and, furthermore, for each query its cacheable expressions. Given the current state of the cache, the best plans chosen by the optimizer for some queries make use of cached results. Changing the state of the cache by admitting a new result and evicting some existing results may completely change the best plan for some queries. The only way to reliably find out would be to re-optimize every query with the new state of the cache, which is a completely impractical proposition. Somehow taking into account the effects of these interactions among cached results is the most difficult part of devising admission and eviction policies for semantic caching. Updates complicate the picture further, because now the lifetime of a cached result becomes dependent on the invalidation rules employed by the cache.

In this study we present a family of cache admission and eviction policies, named *RECYCLE*, specifically designed for semantic caching within a relational database server. We also experimentally compare the effectiveness of our policies with several other caching policies. Our policies were designed to be both effective (high reuse ratio) and efficient (low overhead). The policies need information about the frequency of access to different parts of the database: tables, columns, and subsets of rows. These access statistics are easy to gather and maintain, yet, they offer a solid and flexible framework for dealing with the complexity of the problem. They identify hotspots of the database, i.e., parts of the base data that are frequently accessed by the user queries but are not currently cached. Caching is not automatic, i.e., the decision whether to cache a query result takes into account the current content of the cache and the estimated benefit of the result during its lifetime. These two strategies form the basis of our approach. The contributions of our work are summarized as follows.

- We propose a family of novel algorithms for the problem of semantic caching.
- These algorithms have small computational overhead. Yet they can effectively detect and track the hotspots of the query workloads as they change over time. In addition, the decisions they make take into account the current state of the cache, which is crucial for the performance of a semantic cache.
- We introduce the notion of base data access statistics, and we demonstrate how they can help to tackle a hard problem, namely, rendering our policies aware of the interdependencies of the cache contents, at a small cost.
- An experimental evaluation shows that our algorithms have performance superior to other policies proposed in the literature.

The rest of the paper is organized as follows. We review the relevant literature in Section 2, and we point out why previous approaches are not viable solutions for the general case. In Section 3 we discuss the complexity of the semantic cache management problem, and we outline our approach for solving it. Section 4 introduces some mathematical tools we use later in the paper. In Section 5 we propose the novel *RECYCLE* algorithms for cache

management. An experimental evaluation of our algorithms is presented in Section 6. Finally, we conclude in Section 7, and we discuss some future research directions.

2. Related Work

In this section, we review different cache admission and eviction policies that have been proposed in the literature. We limit the discussion to work related to relational databases and data warehouses.

Least Recently Used (LRU) is probably the most widely used cache replacement policy. Its admission policy is as simple as can be: always admit a referenced object into the cache. Whenever space is needed in the cache, the object whose last reference occurred the furthest in the past is evicted. The LRU policy is not able to discriminate between objects that have frequent references and objects that are rarely accessed. To remedy this problem the LRU-K [9] algorithm was proposed, where eviction decisions are based on the time of the *K-th* to last reference. In practice, LRU-2 provides the best trade-off among performance, storage overhead, and responsiveness to changing access patterns. LRU-based schemes are not suited for a semantic cache because they assume that all objects are of the same size and all have the same replacement cost.

Aggarwal, Wolf and Yu [1] proposed a policy called Size-adjusted LRU (SLRU) that generalizes LRU to objects of varying sizes and varying replacement costs. The application context was caching of (static) web pages. In this context, partial reuse of a cached object does not make sense. However, in semantic caching partial reuse is common and cannot be ignored. Chidlovskii and Borghoff [2] describe a semantic cache designed for caching results of queries to web search engines. Paraphrased in relation terminology, each search engine can be viewed as storing a single table against which a restricted form of select queries (without projection) can be issued. That is, their cached objects correspond to results of select-only queries. A cached object may be partially used by a query and they extend the SLRU algorithm to consider this. However, their extension is rather ad-hoc and specific to their application. Furthermore, their policy ignores differences in replacement cost and does not consider the effects of updates at all.

Stonebraker et al. [12] describe, among other things, a cache management system for Postgres. The context can be paraphrased as follows. Suppose that we have a large number of regular, i.e., not materialized, views defined where the views are typically small. When a query references a view, the view is evaluated completely. To avoid evaluating the same view expression repeatedly, the system may decide to cache the result of some of the views, up to some maximum cache size. The proposed admission and eviction policies take into account several properties of the cached views: size, cost to materialize, cost to access it from the cache, update cost, access frequency, and update frequency. These parameters are combined in a formula that estimates the benefit per unit size (megabyte) of caching a view. For views in the cache, access frequency and update frequency are estimated by maintaining statistics for each cached view. The paper does not describe exactly how these statistics are maintained and whether they adapt to changing access and update patterns. For views not in the cache, the frequencies are estimated as the average frequencies over all the objects in the database. For our application, the major drawback of this scheme is that it does not consider the possibility of evaluating a view, wholly or in part, from other cached views. In addition, a cached view is used only if it is referenced in a query, which does not apply in our case. Furthermore, estimating the frequency of reads and updates by the average over all objects is a very rough approximation. A similar cache management mechanism is used by Keller and Basu [12] but in a different context, namely, semantic caching in a client-server architecture.

WATCHMAN is a semantic cache manager designed by Shim, Scheuermann, and Vingralek. It is targeted for data warehouse application. Their policies include several desired characteristics but, unfortunately, several of their assumptions do not carry over to a more general setting. The initial version, presented in [11], assumes that a cached result is reused only if it matches the query exactly. Their admission and eviction policies are similar to the Postgres policies explained above but updates are not considered. Access frequencies are estimated by keeping track of the last K references to each cached result. Similar to LRU-K, they also retain, for some time, information about results that have been evicted from the cache. In [12], they present an improved version that includes a limited form of partial reuse and considers updates. They consider only queries that are selections over a central fact table joined with one or more dimension tables, normally with a group-by on top. Queries with joins and aggregation, but no selections, are termed data cube queries. Queries that can be rewritten as selections on top of data cube queries are termed slice queries. Their system recognizes two cases of derivability, namely, (a) a data cube is derivable (i.e., can be computed) from a compatible but less aggregated data cube and (b) a slice query is derivable from its underlying data cube query. For all other queries, exact match is required. The system maintains query derivability information in a directed graph. The nodes represent all current cached results, base tables, and some results that are no longer cached but for which statistics are kept. Two nodes are connected by an edge if one node can be derived from the other. The graph is used when selecting victims for eviction. The limited form of derivability considered makes it

possible to easily find the next best alternative for computing a query if a cached result is evicted. However, this is not feasible in a general setting with a more complex derivability structure because the next best alternative depends on the optimizer's reuse rules. Calling the optimizer multiple times to select victims for eviction is clearly too slow and expensive. An additional problem is that their policies completely ignore queries that have never been cached, regardless of how frequently they occur. It also appears that the admission policy is unresponsive to changing hotspots, that is, sufficiently expensive, highly aggregated results will stay in the cache indefinitely even when access patterns change.

The WATCHMAN scheme is used by the DynaMat project [8], which also assumes that the cache manages a restricted set of query results. In particular, the results are either single points in the data space or hyperplanes with each dimension ranging over the whole domain. The experiments indicate that this approach performs better than both LRU and LFU (Least Frequently Used). A cache that stores only a predefined set of query results is also described by Deshpande et al. [4]. Each element in the cache is a *chunk*, a small hyperrectangular subregion of the entire datacube. The chunks are formed in such a way that they are all mutually disjoint. The cache management algorithm admits every new chunk as long as there is free space in the cache. The replacement policy is based on the CLOCK scheme, enhanced with a benefit function, which is proportional to the fraction of the base fact table that a chunk represents. This approach is experimentally shown to perform better than LRU and query caching, where the whole query (and not chunks) is cached. However, the query-caching scheme used in the comparison is rather simplistic. The above algorithm was recently extended [5], so that the benefit function accounts for the fact that a cached chunk can also be used to answer queries at a higher aggregation level.

A very different cache replacement policy is described by Dar et al. [3]. While LRU is based on temporal locality, this work proposes the notion of semantic locality, which attempts to capture the space locality of queries. When using this policy, the cached results that have the greatest semantic distance (Manhattan distance between the centroids of the queries in the data space) from the current query are evicted first. The experimental results show that this approach wins over the simple LRU approach. However, those results were derived by considering a single client accessing a single relation. It is not obvious whether the same techniques will be beneficial in a more general setting.

The above studies propose interesting directions of research, but none of them tackles the problem of semantic cache management in its entirety. Our work tries to remedy this situation, by considering all the relevant parameters under a single framework. The algorithm we propose makes informed decisions both for admission and eviction of query results, based on the characteristics of the query, the access patterns in the workload, and the contents of the cache.

3. Challenges of Semantic Caching

Solving the above problem for a semantic cache is in general more involved than in the case of a traditional cache. The disk pages or relational tuples stored in a traditional cache are mutually exclusive, each one corresponding to a different portion of the base data. Thus, there are no relationships among them that the cache should consider. On the other hand, the objects stored in a semantic cache are not guaranteed to be mutually exclusive. They are the actual answer sets to arbitrary user queries, and they may correspond to overlapping regions of the database. This is exactly why the *semantic* cache problem is harder. Now there are interactions among the *MQRs*. Some *MQR* may be subsumed by one or more other *MQRs*. In this case the former can be derived from the latter, therefore the benefit of storing all of them is questionable. Thus, the benefit of storing a *QR* in the cache is not only a function of the workload characteristics, but is also heavily dependent on the contents of the cache. The benefit of an *MQR* may change as other *MQRs* are added to or deleted from the cache.

As the number of *MQRs* grows, the interactions among them become extremely complex and cumbersome to track and manipulate. Therefore, the problem of selecting the right *QRs* to admit to the cache becomes increasingly hard. Actually, for a reasonably sized cache that holds a number of *MQRs* in the order of several thousands, the aforementioned problem is very expensive to solve, especially since the cache has to operate in real time. Note however, that in some special cases the complexity of this problem is tractable. As has been demonstrated in previous work [5][12], in the restricted environment of a data warehouse the cache is able to monitor the interactions among the *MQRs* with the use of specialized structures. This is made possible because the number of possible interactions is limited by the lattice organization of the data in the warehouse. Though, this is not true for the general case with which we are dealing.

Our strategy for dealing with the problem is twofold. First, we associate the benefit we assign to each *MQR* with the rest of the contents of the cache. Whenever an *MQR* is accessed we compute its benefit as the cost savings resulting from the use of the *MQR* instead of the best other alternative. Obviously, this approach takes into account

the interdependencies of the cached results. However, a careful examination of the aforementioned scheme reveals that it is computationally too expensive to implement. The algorithms we propose work around this problem by providing an efficient and effective estimation of the true cost savings.

Second, we maintain statistics on the frequency of accesses to various base data objects (i.e., tables, columns, etc.). These statistics are very useful, because they give an indication of how popular a new result is going to be if it is cached. By recording only the *base data* accesses we enable our algorithms to make an accurate prediction of what is missing from the cache. The cache should store those query results that cannot be computed from the cache, and that refer to parts of the database that are frequently accessed. Keeping access statistics on the base data gives answer to exactly those questions. This is also a very effective way for giving the admission policy useful hints on the contents of the cache and their interrelationships, which would otherwise be prohibitively expensive to evaluate and take into account.

In the rest of the paper we present the details of our solution. The algorithms we propose are based on the above considerations. They account for the complexity of the problem, yet, in an efficient manner.

4. Tracking Changing Patterns by Exponential Smoothing

To implement our cache management policies we need to be able to compute the weighted averages of a moderately large number of time series, which have the characteristic that most observations are zero. This section explains how exponential smoothing can be used to efficiently maintain such averages. It will become clear in subsequent sections exactly what these time series are as we explain our caching policies.

Let's first consider a single time series, $X=\{X_t, X_{t-1}, X_{t-2}, \ldots\}$. We can compute its weighted average value by a linear combination of the observed values, that is, a function of the form

$$Wavg_t(X) = w_o X_t + w_1 X_{t-1} + w_2 X_{t-2} + \cdots ,$$

where $W = \{w_0, w_1, w_2, \ldots\}$ is a series of weights (constants) whose sum equals one. Normally, the weights would form a non-increasing series to give higher weight to the most recent observations. Moving averages is an example of a widely used estimator. A moving average of length m uses the weights $W = \{1/m, 1/m, \ldots, 1/m, 0, 0, \ldots\}$, that is, the first m weights are equal to 1/m and the remaining weights are all zero.

Exponential smoothing uses exponentially decreasing weights, that is, the average is estimated by the formula

$$Savg_t(X) = \alpha X_t + \alpha(1-\alpha)X_{t-1} + \alpha(1-\alpha)^2 X_{t-2} + \alpha(1-\alpha)^3 X_{t-3} + \cdots ,$$

where α is a constant, $0 \leq \alpha \leq 1$, called the smoothing constant. By subtracting $Savg_{t-1}(X)$ from $Savg_t(X)$, we find that the average at time t can be computed using the formula

$$Savg_t(X) = (1-\alpha)Savg_{t-1}(X) + \alpha X_t ,$$

which is the recursive formula normally used for exponential smoothing. This formula shows one of the key advantages of exponential smoothing, namely, there is no need to store the actual observations to compute the updated average. To compute the average at time t, all we need is the average at time t-1 and the observed value at time t.

Now consider the case when we have n different series that we need to update at time t. For each series X^i, we store its current average $Savg_{t-1}(X^i)$, i=1, 2, …, n. At time t, we receive another set of observations X_t^i, i=1,2, …, n, one for each series and update the n averages using the formula above. This appears to be very efficient but recall that we are interested in the case when most observations are zero, in particular, when there is only one non-zero observation at each point in time. We will now show how the computation can be reorganized so that we only need to update the averages for series receiving non-zero observations.

First, we unwind the recursion and write the average as a function of t previous observations and the average at time t=0. This produces the formula

$$Savg_t(X) = \alpha X_t + \alpha(1-\alpha)X_{t-1} + \alpha(1-\alpha)^2 X_{t-2} + \alpha(1-\alpha)^3 X_{t-3} + \cdots + (1-\alpha)^t Savg_0(X).$$

Next, we multiply both sides of the equation by $(1-\alpha)^{-t}$, which produces the formula

$$(1-\alpha)^{-t} Savg_t(X) = (1-\alpha)^{-t} \alpha X_t + (1-\alpha)^{-t+1} \alpha X_{t-1} + (1-\alpha)^{-t+2} \alpha X_{t-2} + (1-\alpha)^{-t+3} \alpha X_{t-3} + \cdots + Savg_0(X).$$

Now define $W_t = (1-\alpha)^{-t}$, which we will call *smoothing scale factor*, and $CSavg_t(X) = W_t\, Savg_t(X)$. We can then rewrite the series in terms of these quantities

$CSavg_t(X) = W_t \alpha X_t + W_{t-1} \alpha X_{t-1} + W_{t-2} \alpha X_{t-2} + W_{t-3} \alpha X_{t-3} + \cdots + Savg_0(X)$.

Subtracting $CSavg_{t-1}$ from $CSavg_t$, we obtain the following recursive update formulas

$$CSavg_t(X) = CSavg_{t-1}(X) + \alpha W_t X_t, \text{ and}$$

$$W_t = W_{t-1}/(1-\alpha).$$

Notice that if X_t is zero then CSavg remains unchanged. In other words, if we maintain CSavg instead of Savg, we only need to update the average for the series that receives non-zero observations. W_t is the same for all n series so that adds only one additional multiplication to the maintenance procedure. $CSavg_t(X)$ can easily be recovered using the formula $Savg_t(X) = CSavg_t(X)/W_t$.

However, note that W_t and $CSavg_t(X)$ are monotonically increasing, so eventually they will overflow. To avoid overflows, we must periodically rescale all of them. Rescaling is a straightforward operation: for each series X set $CSavg_t(X) = CSavg_t(X)/W_t$ and finally set $W_t = 1$. How often we need to rescale, depends on the value of α and the floating-point representation used. For our application, α would normally be less than 0.05 and the observed values small (less than 100). If double-precision (64-bit) floating-point numbers are used, rescaling every 500 time intervals is sufficient.

The parameter α controls the weight associated with each observation in the time series, and subsequently, the speed with which the procedure forgets old values and adjusts to new ones. Obviously, we are seeking for an α value that will allow us to follow the changing patterns in the series X, but at the same time be insensitive to random, momentary changes. In the experimental evaluation section we explore the sensitivity of our policies when this parameter changes.

We will now explain the procedure of using exponential smoothing to estimate frequencies with an example. Suppose we want to estimate how frequently each of the base tables, $T_1, T_2, \ldots, T_n$, in the database are used in a query. To achieve this we use a vector storing one value, $CSavg(T_i)$, for each table and two additional fields: SumCSavg, storing the sum of all Csavg, and W, the smoothing scale factor. Initially all fields are set to zero, except for W, which is set to one. When a query using, for example, tables T_1 and T_2 is executed, we update the values as follows:

$$CSavg(T_1) = CSavg(T_1) + W \cdot \alpha$$
$$CSavg(T_2) = CSavg(T_2) + W \cdot \alpha$$
$$SumCSavg = SumCSavg + W \cdot \alpha \cdot 2$$
$$W = W/(1-\alpha)$$

The estimated frequency of use of a table T_i can be computed at any time as $CSavg(T_i)/SumCSavg$. It is important to emphasize here that because of the exponential smoothing, these frequencies adapt automatically to changing access patterns. This property allows our policies to effectively work with non-stable query workloads.

5. The *RECYCLE* Algorithms

We now describe the *RECYCLE* cache management policies, and we present the algorithmic tools we employed. Subsequently, we discuss some variations of the basic algorithms.

5.1 Admission and Eviction Policies

Similar to several schemes proposed earlier, the *RECYCLE* admission and eviction policies are based on the notion of *benefit*, more specifically *estimated future benefit*. The benefit is normalized by the size (in megabytes) of the result set. Each *QR* that is candidate for admission and each *MQR* (already in the cache) are associated with a benefit metric. The benefit is a function of the cost to recompute the result of the expression, its frequency of reuse, its update frequency, and its size. As we have already discussed, the benefit should also take into account the current contents of the cache. A candidate *QR* may have a high computation cost from the base data, but very low when the cache contents are employed as well. In that case, the benefit for caching this result should be accordingly low. The detailed algorithms used for estimating the benefit of results already in the cache and candidates for caching are described later in the paper.

We consider for caching only results produced by query expressions consisting of selections and (inner) joins, possibly with a single group-by on top (SPJG expressions). In addition, selection predicates, if present, are limited to conjunctions of range restrictions, that is, conjunctions of predicates of the form *(C_1 op_1 $const_1$ OR C_2 op_2 $const_2$ OR*

...), where C_i is a column, op_i are comparison operators from the set {<, ≤, >, ≥, =, <>}, and $const_i$ are constants. Note that the range restrictions within a conjunct must reference the same column (but different conjuncts may reference different columns). The reason why cached results are limited to SPJG expressions, is merely because our current reuse algorithm (explained in [6]) is also limited to this class of expressions. It is pointless to cache objects that the system will not know how to reuse. The class of cacheable expressions supported may be extended in the future.

Candidates for admission to the cache are identified at the end of query optimization. Not only the final query result but also any SPJG subexpression of the query is a candidate for caching. To find candidates, we traverse the final query plan and identify cacheable expressions, that is, SPJG expressions with selection predicates, if any, that are range restrictions. We compute the estimated benefit for each cacheable expression and select the one with the highest benefit as a candidate for caching. That is, for each query we select at most one expression for caching.

The actual caching happens during execution of a query plan so the final decision whether to admit a *QR* must be made during execution time. Most systems cache query plans and reuse them if the same query is submitted again by the same or other users. Hence, a query plan may be executed multiple times. We must reassess whether to admit *QR* to the cache every time the plan is executed because the contents of the cache may have changed. If the *QR* already exists in the cache, there is no need to cache it again. If it does not exist in the cache, it is admitted only if its estimated benefit exceeds the estimated benefit of the *MQRs* that it evicts. In order not to slow down execution, it is important that the final decision whether to admit a *QR* into the cache and what *MQRs* to evict can be made quickly.

Now consider the problem of choosing one or more victims for eviction. This problem turns out to be equivalent to the standard knapsack problem: among the *MQRs* already in the cache and the candidate for admission, select the subset that maximizes the total benefit and fits in the space limit of the cache. The total benefit is computed as the sum of the benefit (which is normalized by the size) times the size in megabytes of each *MQR* and of the candidate for admission. As is well known, the knapsack problem is NP-complete but there are good approximate algorithms [20]. The simplest algorithm is the greedy heuristic: sort the items in decreasing order of "price per pound" and then pick items from the beginning of the list until the knapsack is full. As we show in the Appendix, the approximation that the greedy heuristic achieves in our context is close to optimal. In our case, "price per pound" translates into benefit per megabyte. We use the dual version of the greedy heuristic; namely, evict *MQRs* from the cache in increasing order of their benefit per megabyte. This heuristic gives us a list of victims but we still need to decide whether exchanging them for the candidate *QR* will increase the total benefit. The total benefit of an *MQR* or a candidate *QR* is its benefit per megabyte times its size in megabytes. We add up the total benefits of the selected victims and compare the sum to the total benefit of the candidate *QR*. The *QR* is admitted (and the selected victims evicted) only if its total benefit is higher than the sum of the total benefits of the victims.

The *RECYCLE* cache management policies are summarized as follows.

Admission policy: After optimization of a query, traverse the final plan and identify the cacheable subexpression *QR* with the highest estimated benefit per megabyte. If the estimated benefit of *QR* is positive, mark *QR* as a candidate for caching during execution. Before execution of a query plan with a marked subexpression *QR* begins, decide whether to admit *QR* to the cache. If *QR* is already in the cache (cached by a previous execution of the plan), do not admit *QR* again. Otherwise, consider whether there is enough free space for *QR* in the cache. If there is enough space, then admit *QR*. If there is not enough space, then admit *QR* only if its estimated total benefit is higher than the sum of the total benefits of the *MQRs* that must be evicted in order to free up the space needed by *QR*.

Eviction policy: Repeatedly evict the *MQR* with the lowest estimated benefit among all the *MQRs* remaining in the cache until enough space has been freed up.

5.2 Estimating the Benefit of a Cached Result

The basic idea is to credit an *MQR* with the benefit, i.e., the cost saving, attributable to it whenever it is used during query execution. For each *MQR*, we keep track of its average benefit per megabyte per query executed. When a query *Q* is executed and a set of *MQRs* are used in order to produce the answer, we would like to credit each MQR_i used with a benefit computed as

$$B_{MQRi}(Q) = (C_Q(without\ MQR_i) - C_Q(with\ MQR_i))/|MQR_i|.$$

The components in the formula are as follows.

- $C_Q(without\ MQR_i)$ is the cost to compute query *Q* assuming that the MQR_i is not cached;
- $C_Q(with\ MQR_i)$ is the current cost to compute query *Q*, that is, making use of MQR_i;
- $|MQR|$ is the size of MQR_i (in megabytes).

This formula has the advantage that the benefit attributed to an *MQR* takes into account the current state of the cache. Now consider whether and how we can obtain the values needed in the formula. The size, $|MQR_i|$, is readily available, since we know the amount of space occupied by each cached result. For $C_Q(with\ MQR_i)$ we can use the optimizer's cost estimate for query *Q*. However, $C_Q(without\ MQR_i)$ is not readily available. To obtain an estimate we would have to mark MQR_i as unavailable and reoptimize the query. Reoptimizing once for each *MQR* used in a query is clearly too expensive, thus we have to come up with a solution that does not require reoptimization.

An alternative approach is the following. Query *Q* reads some fraction of the rows in MQR_i. We know the cost of computing the cached result when it was created and we can also estimate the cost of reading the cached result. If we assume that the query would have recomputed the part of the cached result that it read, it is reasonable to estimate the benefit using the following formula

$$B_{MQRi}(Q) = F_Q \cdot (C(compute\ MQR_i) - C(read\ MQR_i))/|MQR_i|,$$

where

- F_Q is the fraction of MQR_i read by query *Q*,
- $C(compute\ MQR_i)$ is the cost of computing the result of MQR_i (excluding the cost of storing it), and
- $C(read\ MQR_i)$ is the cost of reading MQR_i.

This formula does not require any reoptimization. The only quantity in the formula that depends on the current query *Q* is F_Q. An estimate of the fraction of the cached result read by the query can be obtained easily be examining the execution plan of the query (our solution) or by counting rows read during query execution. For $C(compute\ MQR_i)$ we simply use the optimizer's cost estimate at that node in the plan. We estimate $C(read\ MQR_i)$ using the optimizer's normal cost formulas and its estimates of row count and average row size for the result.

This formula takes into account the state of the cache when MQR_i was created. MQR_i may have been computed using base tables only or using some other cached results. This will be reflected in the cost, that is, if MQR_i was computed using other cached results, $C(compute\ MQR_i)$ will be lower than if it had been computed directly from base tables. However, the benefit does not reflect the current cost of computing MQR_i, that is, the benefit remains the same even if the cached results used to compute MQR_i are evicted from the cache. This could be remedied by tracking which other cached results MQR_i depends on. When one of the cached results on which MQR_i depends on is evicted, we reoptimize (but do not recompute) the query expression defining MQR_i to get a new estimate for $C(compute\ MQR_i)$. This increases the cost of eviction because evicting a cached result triggers reoptimization of all other cached results that depend on it. To avoid a flurry of reoptimization, we could simply mark the affected cached results as needing reoptimization and defer the actual reoptimization until the next time the cached result is used. Further investigation of this idea is needed because it is not immediately clear under what circumstances the extra cost and complexity of dependency tracking and reoptimization are justified. Note that we suggest calling the optimizer *after* the eviction decision has been made to fix up future estimates but not in order to make the eviction decision. We have not implemented this idea.

We keep track of the average benefit for each *MQR* in the cache using the low-overhead version of exponential smoothing presented in the previous section. Each *MQR* in the cache has an associated data structure, called its descriptor, that contains among other things four fields needed for benefit estimation: (cumulative) average benefit, *CumBenefit*; cost of computing the result, *CostCompute*; cost of reading the result, *CostRead*; and size of the result (in megabytes), *SizeMB*. The fields *CostCompute*, *CostRead* and *SizeMB* are initialized when the result is admitted to the cache. *CumBenefit* is initialized to zero. In addition, there is one field, *ScaleFactor*, common to all cached results that keeps track of the current smoothing scale factor. When a query is executed, *CumBenefit* is updated for each *MQR* used in the query according to the formula

$$CumBenefit = CumBeneft + \alpha \cdot Scalefactor \cdot f_Q \cdot (CostCompute - CostRead)/SizeMB.$$

Finally, the global smoothing scale factor is updated acccording to the formula

$$Scalefactor = Scalefactor/(1-\alpha).$$

The *MQRs* (actually the *MQR* descriptors) are organized in a heap structure to enable fast access to the *MQRs* with the lowest average benefit.

5.3 Estimating the Benefit of a Candidate for Admission

When a new query result *QR* is considered for admission to the cache, we need to compute its expected benefit. The query result may be the final result of a user query or the result of a subexpression of the query. The benefit

depends on how frequently it is likely to be accessed in the future, the cost savings from caching it as opposed to recomputing it, its size, and how long the result is expected to remain valid. These parameters are combined in a function that represents the benefit of caching *QR* per unit size and per unit time:

$$B_{QR} = (P_{QR} \cdot F_{QR}(C(compute\ QR) - C(read\ QR)) - V_{QR} \cdot C(write\ QR)\ )/|QR|$$

The parameters used in the formula are as follows.

- P_{QR} is the probability of a (future) query using the result if cached;
- F_{QR} is the average fraction of the result read by a query that uses the result;
- *C(compute QR)* is the current cost of computing *QR*, regardless of whether it was computed entirely from base tables or from some existing *MQRs*;
- *C(read QR)* is the cost of reading *QR* from the cache;
- V_{QR} is the time *QR* is expected to remain valid;
- *C(write QR)* is the cost of materializing (writing) *QR*;
- |*QR*| is the size of *QR* (in megabytes).

We already discussed how to compute *C(compute QR)* and *C(read QR)* in the previous section. *C(write QR)* can be computed in the same way as *C(read QR)*, namely, by using the optimizer's normal cost formulas for estimating the cost of writing a temporary result.

P_{QR} is the probability of *QR* being used by a future query as perceived by the *base data*. We are interested in how frequently the system accesses base tables for answering a query, because those are the queries that may use this result if it is cached. This way of calculating P_{QR} is very important, giving our caching policy the desirable property that results covering "unpopular" parts of the database are unlikely to be cached. It does, however, ignore the fact that some queries that used other cached results before may now use *QR* instead (i.e., once *QR* is cached).

F_{QR} is the average fraction of the result read by a query that uses the result. This quantity is rather difficult to estimate; it depends both on the characteristics of the cached result and the characteristics of the query. As a practical solution, we estimate this by the average observed fraction across all *MQRs* kept in the cache. That is, it is the exponentially smoothed average of the usage fractions F_Q (defined in the previous section) observed as queries are executed. One could probably do better by dividing the *MQRs* into classes based on some characteristics, for example, tables and columns used, and estimating F_{QR} separately for each class. However, we have not investigated this issue in detail.

V_{QR} is the time *QR* is expected to remain valid considering the invalidation rules used by the cache. As a base table *T* is updated, some of the cached results containing data from *T* may no longer be correct and have to be purged from the cache. Exactly when a cached result is declared invalid depends on the invalidation rules used. In principle, we can always decide correctly whether to invalidate a cache result by recomputing the expression and comparing the old and new results. This is a totally impractical rule, of course, so in practice we have to use more efficient, and therefore more conservative rules. The simplest rule is to invalidate an *MQR* whenever one of its base tables is updated. A slightly better rule would also consider what columns were affected: if the expression defining an *MQR* does not reference any of the updated columns, it cannot be affected by the update, and remains valid. Estimating V_{QR} requires that we maintain some statistics on update frequencies. This issue will be discussed further in Section **Error! Reference source not found.**, when we discuss access statistics.

Note that the benefit formula implicitly takes into account the cache contents. If *QR* was computed cheaply from some *MQRs* already in the cache, *C(compute QR)* will be low and the cost savings small. Even if computing *QR* directly from base tables is high, the actual cost and the potential cost savings may be small. This gives our caching policy the property that a result is unlikely to be cached if it covers queries over a part of the database that is already well covered by other *MQRs* in the cache. In that case, little is to be gained by caching *QR*.

The *RECYCLE* cache management policies are designed to allow in the cache only those results that are frequently accessed and that result in substantial savings for future queries. This is true even when the characteristics of the workload change over time.

5.4 Estimating the Reuse Frequency of a Candidate for Admission

To use the benefit formula described in the previous section we need to estimate the probability of reuse of candidates for admission. In this section, we discuss how we compute the probability of reuse from a collection of more elementary probabilities. In the next section, we show how these more elementary probabilities can be efficiently estimated by maintaining certain simple access statistics on base data.

QRs that are candidates for caching are defined by SPJG expressions of the following general form:

```
SELECT <List of output expressions>
FROM   R1, R2, ..., Rn
WHERE  <RangePredicate1> AND <RangePredicate1> AND ...
       AND <JoinPredicate1> AND <JoinPredicate2> AND ...
[GROUP BY <List of grouping expressions>]
```

where each <RangePredicate> is a range restriction over some column in one of the source tables and each <JoinPredicate> is a predicate involving columns from more than one table. The output expressions and grouping expressions are typically plain column references, but may also be arbitrary scalar expressions.

Given a *QR* of this type, we want to estimate the probability that a (future) query will use the result of this expression. This probability depends on a variety of factors, including the characteristics of the expression, the current contents of the cache, and the algorithm for recognizing whether a result can be used in a query. Clearly, we have to make some simplifying assumptions. The first assumption is one of independence, namely, source tables are selected independently of each other, output columns are selected independently of each other, columns on which range predicates are applied are selected independently of each other, applicable join predicates are selected independently of each other, and grouping columns are selected independently of each other. Second, we assume that a cached result can only be used to answer a query expression that references the same tables. These are admittedly strong assumptions, yet they allow us to efficiently tackle a hard problem and still deliver superior performance.

Suppose we have available or can estimate the following probabilities:

1. $P_{tab}(R_i)$, the probability that a query uses table R_i as a source table;
2. $P_{out}(R_i.C_j \mid R_i)$, the probability that column C_j of table R_i is used as an output column given that table R_i occurs as a source table;
3. $P_{sel}(R_i.C_j \mid R_i)$, the probability that a query has a range restriction on column C_j of table R_i that is contained in the corresponding range restriction of the *QR*, again given that table R_i occurs as a source table;
4. $P_{join}(\ joinpred(R_i, R_j) \mid R_i \ \& \ R_j)$, the probability that a query uses the same join predicate between tables R_i and R_j as the one used in the *QR*, given that R_i and R_j occur as source tables;
5. $P_{group}(R_i.C_j \mid R_i)$, the probability that a query uses column C_j of table R_i as a grouping column, given that table R_i occurs as a source table;

Assuming tables are selected independently, the probability of a query using tables $R_1, R_2, \ldots, R_n$ is then

$$P^{tot}_{tab} = \prod_{i=1}^{n} P_{tab}(R_i) \cdot$$

Let $Col(R_i)$ denote the set of columns of table R_i and $Cout(R_i)$ the set of columns of table R_i output by *QR*. The *QR* is usable by a query only if the query outputs a subset of the columns output by *QR*, that is, if it outputs no columns not output by *QR*. This probability can be computed as

$$P^{tot}_{out} = \prod_{i=1}^{n} \prod_{j} (1 - P_{out}(R_i.C_j \mid R_i)),$$

where j ranges over all columns in $Col(R_i) - Cout(R_i)$, that is, all columns of R_i not output by *QR*.

A query cannot be computed from the *QR* unless all its range restrictions are contained in the corresponding range restrictions of the *QR*. This probability can be computed as

$$P^{tot}_{sel} = \prod_{i=1}^{n} \prod_{j} P_{sel}(R_i.C_j \mid R_i),$$

where j ranges over all columns that are range restricted in *QR*.

A query cannot be computed from *QR* unless it contains, at least, all the join predicates contained in *QR*. The probability of this occurring can be computed as

$$P^{tot}_{join} = \prod_{i=1}^{n} \prod_{j=1}^{n} P_{join}(join(R_i, R_j) \mid R_i \,\&\, R_j).$$

If QR contains a grouping clause, a query cannot be computed from QR unless its grouping clause specifies a subset of the grouping columns in QR. In other words, its grouping list cannot contain any columns not in the grouping list of QR. Let $Cgroup(R_i)$ denote the set of grouping columns from table R_i used in QR. This probability can be computed as

$$P_{group}^{tot} = \prod_{i=1}^{n} \prod_{j} \left(1 - P_{group}(R_i.C_j \mid R_i)\right),$$

where j ranges over all columns in $Col(R_i) - Cgroup(R_i)$, that is, all columns of R_i that are not used as grouping columns in QR.

Finally, we combine all the above probabilities to get the probability of reuse of the QR:

$$P_{reuse} = P_{tab}^{tot} \cdot P_{out}^{tot} \cdot P_{sel}^{tot} \cdot P_{join}^{tot} \cdot P_{group}^{tot} .$$

5.5 Access Statistics for Base Data

In this section, we describe a mechanism for keeping track of the frequency of accesses to various base data objects (i.e., tables, columns, etc.), which we use in our cache management policies. We use these statistics to estimate how popular a new result is expected to be if it is cached, which then affects the expected savings from caching the result. Note that we are concerned only with accesses to base tables and not with accesses to cached results. Expected savings for results already cached are estimated by a different mechanism.

Obviously, the finer granularity we use in the statistics, the more accurate estimates we get. However, accuracy comes at the expense of additional CPU and storage overhead. We choose to monitor the frequency of accesses to the following objects:

1. Tables;
2. Joins between two tables;
3. Output columns of tables;
4. Selection ranges for columns (i.e., column access histograms);
5. Sets of grouping columns.

We use exponential smoothing to ensure that our frequency estimates adapt to changing patterns in the workload. This also allows us to update the estimates very efficiently. We also keep the frequencies normalized at each step so they can be treated directly as probabilities.

We'll explain the procedure of access statistics maintenance in detail for tables, and then briefly cover other objects. We maintain one statistics object for each base table (if the base table has been used in at least one query). The table statistics object contains just one field, *UnscaledFreq*, which is initialized to one. In addition, there are two global variables. The smoothing scale factor, *ScaleFactor*, and *SumUnscaledFreq*, which stores the current sum of the all the existing *UnscaledFreq* fields. The *ScaleFactor* field is *unique* and common to all the statistics objects we keep for a database. It is updated every time a query is executed as follows

$ScaleFactor = ScaleFactor/(1-\alpha).$

Now assume that a query accessing base tables $R_1, \ldots, R_k$ is executed. Then, *ScaleFactor* is updated as above, and the for the rest of the variables we apply the formulas:

$UnscaledFreq(R_i) = UnscaledFreq(R_i) + \alpha \cdot ScaleFactor, 1 \le i \le k,$ and

$SumUnscaledFreq = SumUnscaledFreq + \alpha \cdot ScaleFactor \cdot k.$

The probability of a query accessing table R_i, $P_{tab}(R_i)$, can be estimated at any time by normalizing the frequency: $P_{tab}(R_i) = UnscaledFreq(R_i)/ScaleFactor$. According to the formula presented in Section 5.4 for computing the probability of a query accessing together all the tables it references, we can estimate P_{tab}^{tot} simply as $P_{tab}^{tot} = P_{tab}(R_i) \cdots P_{tab}(R_k)$.

Following a procedure similar to the one outlined above we can also compute the probabilities with which a query references a join between two tables, a set of output columns, or a set of grouping columns. In the case where the query involves range restrictions on one or more columns we follow a slightly different approach. This is necessary in order to correctly estimate the probability of *reuse* of such a query result. Our strategy is outlined in the next section.

5.5.1 Access Histograms for Column Selection Ranges

In order to be able to make accurate estimates of access frequencies to a column subset we construct and maintain *Access Histograms (AccHist)* on individual columns. These histograms divide the active domain of the column to a prespecified number of buckets n_B. Initially, this subdivision of the domain into buckets is done in a uniform way, assigning an equal number of domain values to each bucket (similar to equi-width selectivity histograms). We associate every column with two histograms, *AccHistGreater* and *AccHistLess*, and each bucket of these histograms stores an *UnscaledFreq* field. These fields are initialized according to the uniform distribution, that is, $UnscaledFreq = 1/n_B$, unless we have a reason to favor some of them. If, for example, some column stores values related to time, and we know that the majority of the queries ask for recent values, then we may tailor the initial distribution accordingly. Note though, that even if we start with the uniformity assumption, the exponential smoothing technique will cause the *UnscaledFreq* values to asymptotically converge to the real distribution (under the assumption that the query workload exhibits some reference locality). Two more variables, *SumUnscaledFreqG* and *SumUnscaledFreqL*, keep for each of the two histograms the current sum of all the *UnscaledFreq* values. Finally, we also use *ScaleFactor*, which is unique and defined for the entire database.

The *AccHistGreater* histogram is updated in response to selection conditions of the form *R.C opGreater const*, where *C* is the name of a column of table *R*, $opGreater \in \{>, \geq\}$, and *const* is a value in the active domain of *C*. Similarly, *AccHistLess* is activated by selection conditions of the form *R.C opLess const*, where $opLess \in \{<, \leq\}$. Queries that specify a closed range restriction within the active domain of *C*, i.e., with selection conditions of the form $const_1$ *op R.C op* $const_2$, are divided into two parts each one involving a single comparison operator, and are treated separately. Assume that a query *Q* specifies *k* range restrictions with *opGreater* operators, and *l* restrictions with *opLess* operators is executed. For each restriction we locate the bucket *b* that contains *const* and update its *UnscaledFreq* field. We update the statistics as follows

$$ScaleFactor = ScaleFactor/(1-\alpha),$$

$$UnscaledFreqG(b_i) = UnscaledFreqG(b_i) + \alpha \cdot ScaleFactor,\ 1 \leq i \leq k,$$

$$UnscaledFreqL(b_i) = UnscaledFreqL(b_i) + \alpha \cdot ScaleFactor,\ 1 \leq i \leq l,$$

$$SumUnscaledFreqG = SumUnscaledFreqG + \alpha \cdot ScaleFactor \cdot k, \text{ and}$$

$$SumUnscaledFreqL = SumUnscaledFreqL + \alpha \cdot ScaleFactor \cdot l.$$

Remember that *ScaleFactor* is updated only once for each query that is executed.

Now we are ready to estimate the probability P_{sel} that a range restriction can be reused by other queries. Consider a single selection condition of query *Q* of the form $R_q.C_q\ opGreater_q\ const_q$, and let b_q be the bucket in which $const_q$ lies. Bucket b_q alone can give us an estimate of the probability of a query having a similar restriction, that is, $R_q.C_q\ opGreater_q\ const_2$, where $const_2$ lies in the range of values of bucket b_q. However, any query with a restriction $R_q.C_q\ opGreater_q\ const_3$, where $const_3 \geq const_q$, will be able to reuse the result of the selection condition, albeit only a portion of it. Thus, the probability $P_{sel}(R_q.C_q/ R_q)$ is calculated as follows

$$P_{sel}(R_q.C_q \mid R_q) = \frac{\sum_{i=b_q}^{n_B} UnscaledFreqG(b_i)}{SumUnscaledFreqG},$$

where n_B is the total number of buckets of *AccHistGreater*. Analogously, for range restrictions involving *opLess* comparison operators we have

$$P_{sel}(R_q.C_q \mid R_q) = \frac{\sum_{i=1}^{b_q} UnscaledFreqL(b_i)}{SumUnscaledFreqL}.$$

It is likely that the selected range will not align exactly with the bucket boundaries, but will rather intersect a bucket at an arbitrary point. In that case, we can use linear interpolation in order to refine the estimates we get for the *UnscaledFreq* field.

In the last step of the procedure, following the formula we presented in Section 5.4, we can derive the probability of reuse of the range restrictions of query *Q* by multiplying the individual P_{sel} values for all the $k+l$ range restrictions.

So far, the implicit assumption in our discussion was that the column under consideration is a numeric attribute. However, we can apply the same techniques for a categorical attribute as well, if there exists an ordering function on that attribute.

Evidently, the choice we made to construct the access histograms using buckets of equal width is not the optimal solution. Thus, we need a way to change the bucket boundaries in order to fit better the underlying access distribution. This procedure cannot be performed just once at the beginning, but should rather be dynamic, because the access distribution is evolving along with the query workload. There are several approaches that have been proposed in the literature for dynamically managing histograms (albeit, selectivity histograms) [14][16][15]. This problem is very interesting in out setting, but we do not pursue it any further in the current work.

5.6 Accounting for Updates in the Base Data

An important issue that we have not discussed so far is that of updates in the base data. When the contents of the relations in the database change, the cache must change as well in order to reflect the latest state of the database. When propagating the changes from the database to the cache we first have to identify the affected *MQRs*. Then there are three alternatives on how to enforce consistency between the database and the cached results. The first is to recompute the affected *MQRs*; the second is to incrementally maintain them; and the third is to invalidate them. Obviously, regardless of the choice we make, caching a query result that will soon be updated is a waste of resources, and we would like to avoid it.

In order to remedy this situation we keep track of the frequency of updates to the same database objects as for the access statistics, namely, tables, joins between tables, output columns of tables, selection ranges for columns, and sets of grouping columns. Once again, we use exponential smoothing in order to take into account changing patterns in the workload. Then, we just have to penalize the benefit estimate of a candidate for admission according to the frequency with which is gets updated.

5.7 Revisiting the Query Execution Cycle

In order to implement the *RECYCLE* cache management strategies we have to make some minor changes to the query execution cycle. When a query is submitted to the system it first enters the optimizer, where the query is optimized. The output is the estimated best execution plan, which may reference both cached objects and base data. It is this plan that is subsequently executed by the execution engine in order to produce the result of the query. The query execution plan may also be cached to avoid the reoptimization if the same query is encountered again.

The final query execution plan is delivered to the cache subsystem for preprocessing. The cacheable subexpresssions are identified and the one with the highest estimated benefit is selected and marked. The cache gathers all the essential statistics and information in a succinct representation, and attaches this data packet to the plan. The information in this packet is everything that is needed for computing the benefit functions, as well as updating the base data access statistics. The data packet that has been attached to the query execution plan travels with it until execution time. When execution begins, the data packet is handed over to the cache. It is then that the actual processing takes place. The cache updates all the pertinent statistics, computes the new benefit functions and updates the old ones, makes the decision of whether to cache the new result or not, and if necessary which old results to evict from the cache. All the relevant information is preprocessed, so the computation overhead during execution time is minimal.

We consciously made the decision to leave for execution time as little processing as possible because a query is optimized only once but may be executed several times. Even if the state of the system has changed since the query was optimized, the effort is not wasted, because it is the same plan that will be executed. The only data that *may* be outdated are the cost estimates used in the benefit functions. If in the meantime the state of the cache has changed in such a way that the computation of the *MQRs* involved in the query is affected, then the cost savings estimate will not be accurate. Nevertheless, we expect this situation to have only a small effect on the performance of the cache, and we believe that the advantages of our approach justify the current choices.

5.8 Variations of the *RECYCLE* Algorithms

There are several possible variations of the policies we presented, with different levels of computation time and space costs. An obvious question is how much of the access statistics is enough. However, the answer to this question is not straightforward since it heavily depends on the characteristics of the workload, and can vary greatly. There is a tradeoff between the accuracy of the statistics and the complexity and performance (time-wise) of the system. Along this direction, we could drop some or all of the base data access statistics. In the case where we choose not to keep any access statistics, we could replace P_{QR} in the benefit function for admissions with the average access frequency over all the *MQRs*. In effect, we are replacing the estimate of the future access frequency for a new

MQR with the average access frequency of all *MQRs*. We call this policy *RECYCLE-s* (*RECYCLE* simplified), and we study its performance in the experimental evaluation.

We also experiment with another variation of the policies where we set $P_{QR} = 1$. In this case we ignore the P_{QR} factor altogether. The underlying assumption is that all the candidates for admission to the cache are equally likely to be accessed again in the future, and therefore are assigned the same weight with respect to that parameter. In the experimental section we refer to this policy as *RECYCLE-c* (*RECYCLE* constant).

6. Experimental Evaluation

In order to evaluate the performance of the proposed cache management policies we implemented the algorithms described in this paper and ran a series of experiments. The implementation was done in the SQL Server commercial database management system, written in C++, for the Windows 2000 operating system. For the experiments we used a synthetic dataset based on the TPCD benchmark, and a synthetically generated workload over this dataset. The size of the dataset is 100MB. The queries in the workload were generated following a specified distribution over the elements of the dataset (i.e., relations, attributes, ranges of attributes). In the workload we use for the experiments we have specified a number of hot spots. That is, some regions of the dataset are accessed more often that others. In the workload there are a total of 500 queries. The measure we report in the experimental graphs is the *cost savings*, which indicates how much of the total execution cost of the queries in the workload was saved because of the use of the cache. We set the size of the cache to 50MB.

6.1 Changing the Smoothing Constant

In the first set of experiments we only examine the *RECYCLE* policies. More specifically, we are interested in evaluating the effect of changing the smoothing constant. Remember that we apply an exponential smoothing mechanism over both the access statistics for the base data and the benefit values of the cached results. We refer to the smoothing constant for the base data statistics as *alpha-b*, and the one for the benefit of *MQRs* as *alpha-c*. The smoothing constant controls the speed with which the algorithm forgets the old values and adjusts to the new ones. When the value of the smoothing constant is high then the procedure adjusts to the new values faster and consequently has shorter memory.

Figure 1 depicts the effect on cost savings of varying parameter *alpha-b* when *alpha-c* is constant, for three different values of *alpha-c*, namely 0.2, 0.1, and 0.05. As we increase the value of *alpha-b* the cost savings decrease or remain constant. We observe the same trend across all values of *alpha-c*.

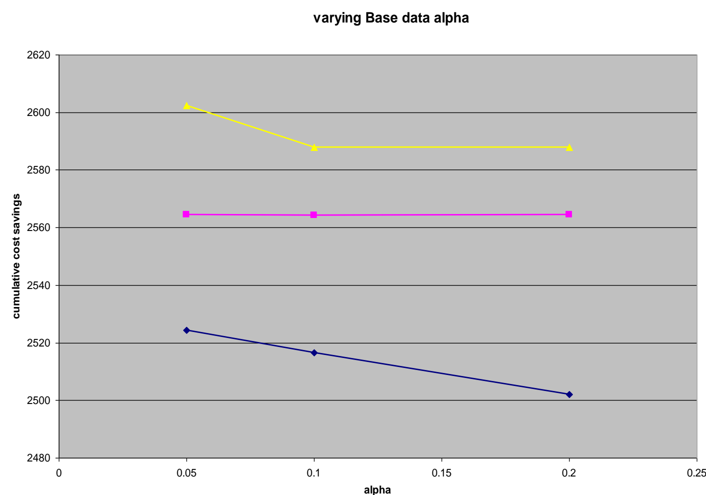

Figure 1: Varying the parameter alpha of the statistics on the accesses to the base data.

In Figure 2 we show the same experiments as before, but this time we vary the parameter *alpha-c* for three constant values of *alpha-b*. Similarly to the previous graph we observe that the cost savings are diminishing when we increase the value of *alpha-c*. This means that in both cases it is beneficial to keep the values of the smoothing constants low, so that the cache management policies follow the changing patterns of the workload and remain

unaffected by the random changes. It is also evident by examining the two figures that the *RECYCLE* policies are more sensitive on the *alpha-c* parameter that the *alpha-b*.

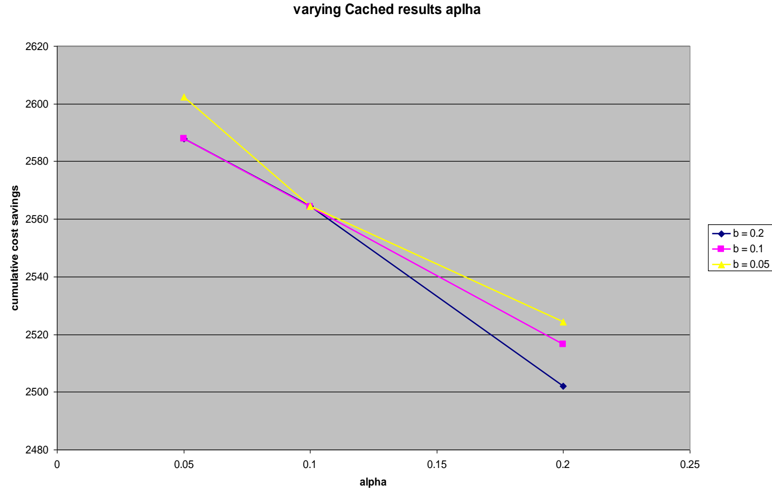

Figure 2: Varying the parameter alpha of the statistics on the cached results.

6.2 Comparison to Other Cache Management Policies

In order to assess the performance of our approach we compare the *RECYCLE* policies with other cache management policies proposed in the literature. In the following we briefly describe the alternatives we consider.

- **Random**: Always admit a new result. Randomly evict cached results to make space for the new admission.
- **Least Recently Used (LRU)**: Always admit a new result. Evict the cached result that was least recently used.
- **Least Frequently Used (LFU)**: Always admit a new result. Evict the cached result that is least frequently used. When computing the frequency of use of the cached results we also apply an exponential smoothing mechanism.
- **Fixed Cost With No Aging**: A benefit value is associated to each candidate for admission based on the ratio (cost of computing the result)/(size of the result). A new result is admitted if its benefit is greater than the total benefit of all the cached results that have to be evicted in order to make space for this new result. The cached results with the lowest benefit values are evicted.
- **Fixed Cost**: This policy is the same as the previous one, with the added feature that an exponential smoothing mechanism is used, which makes cached results not recently used more attractive for eviction.
- **Infinite**: Always admit a new result. Never evict any cached result.

The only reason we include the random and infinite policies in our evaluation is as a reference for the performance of the rest of the algorithms. We expect all the other policies to perform better than random and worse than infinite. Evidently, the infinite policy is not realistic, because it assumes that the available space for the cache is unbounded. Note however, that infinite is not the same as optimal. The optimal algorithm must have a bound on the size of the cache, and because of this is expected to perform at most as good as infinite. This implies that the upper bound for the performance of the cache management policies is probably lower than what indicated by infinite. We did not implement the optimal policy, because it is an NP-hard problem, and the exponential growth of the solution space was prohibitive for the size of the problems we considered in the experiments.

Figure 3 depicts the cumulative cost savings for the workload of the 500 queries and for all the aforementioned cache management policies. As expected, the performance of all the algorithms is between infinite and random. The *RECYCLE* policy is performing well above the competitors, closely following the infinite. This experiment demonstrates the superiority of our solution to the cache management problem.

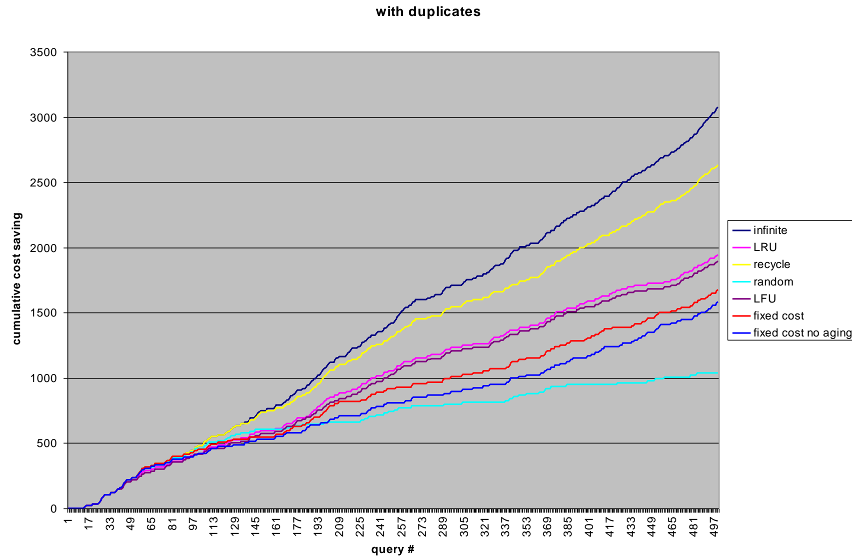

Figure 3: Cumulative cost savings for various cache management policies.

In the last set of experiments we investigate the performance of the variations of the *RECYCLE* policies. The graph of Figure 4 depicts the cumulative cost savings for the query workload and for each one of the infinite, *RECYCLE*, *RECYCLE-s*, and *RECYCLE-c* policies (described in Section 5.8). In this case we want to assess the relative importance of various parameters in our approach.

The results of the experiments show that we have to pay a significant performance penalty for completely ignoring the access statistics to the base data (*RECYCLE-c*). Even when trying to approximate this parameter, as in the case of *RECYCLE-s*, we do not get performance close to the one achieved by the *RECYCLE* policy. This is a strong argument in favor of our decision to employ the access statistics to base data, so that the cache management policies can take more informed actions.

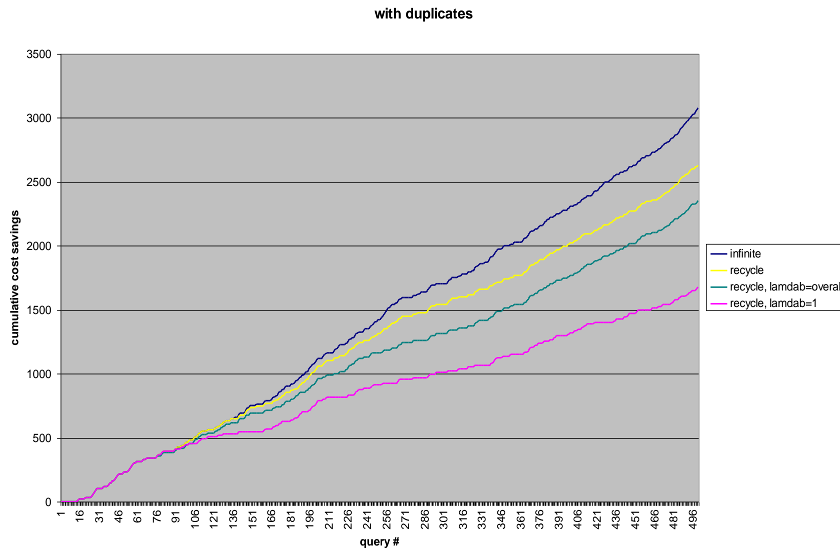

Figure 4: Cumulative cost savings for different variations of the RECYCLE cache management policies.

7. Conclusions and Future Work

In this paper we propose a family of novel cache admission and eviction policies, named *RECYCLE*, specifically designed for semantic caching within a relational database server. These algorithms have small computational overhead. Yet they can effectively detect and track the hotspots of the query workloads as they change over time. In addition, the decisions they make take into account the current state of the cache, which is crucial for the

performance of a semantic cache. We introduce the notion of base data access statistics, and we demonstrate how they can help to tackle a hard problem, namely, rendering our policies aware of the interdependencies of the cache contents, at a small cost. Caching is not automatic, i.e., the decision whether to cache a query result takes into account the current content of the cache and the estimated benefit of the result during its lifetime. These two strategies form the basis of our approach. An experimental evaluation shows that our algorithms have performance superior to other policies proposed in the literature.

8. References

[1] C. Aggarwal, J.L. Wolf, and P.S. Yu: Caching on the World Wide Web, TKDE 11(1), 1999:94-107.

[2] Boris Chidlovskii, Uwe M. Borghoff: Semantic Caching of Web Queries. VLDB Journal 9(1), 2000:2-17.

[3] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, Michael Tan: Semantic Data Caching and Replacement. VLDB 1996: 330-341

[4] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, Jeffrey F. Naughton: Caching Multidimensional Queries Using Chunks. SIGMOD 1998: 259-270

[5] Prasad Deshpande, Jeffrey F. Naughton: Aggregate Aware Caching for Multi-Dimensional Queries. EDBT 2000: 167-182

[6] Jonathan Goldstein, Per-Åke Larson: Optimizing Queries Using Materialized Views: A practical, scalable solution. SIGMOD 2001 (to appear).

[7] Arthur M. Keller, Julie Basu: A Predicate-based Caching Scheme for Client-Server Database Architectures. VLDB Journal 5(1): 35-47 (1996)

[8] Yannis Kotidis, Nick Roussopoulos: DynaMat: A Dynamic View Management System for Data Warehouses. SIGMOD 1999: 371-382

[9] Martello, S. and Toth, P., Knapsack Problems: Algorithms and Computer Implementations, John Wiley, 1990.

[10] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum: The LRU-K Page Replacement Algorithm for Database Disk Buffering. SIGMOD Conference 1993: 297-306

[11] Peter Scheuermann, Junho Shim, Radek Vingralek: WATCHMAN: A Data Warehouse Intelligent Cache Manager. VLDB 1996: 51-62

[12] Junho Shim, Peter Scheuermann, Radek Vingralek: Dynamic Caching of Query Results for Decision Support Systems. SSDBM 1999: 254-263

[13] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, Spyros Potamianos: On Rules, Procedures, Caching and Views in Data Base Systems. SIGMOD 1990: 281-290

[14] Ashraf Aboulnaga, Surajit Chaudhuri: Self-tuning Histograms: Building Histograms Without Looking at Data. SIGMOD 1999: 181-192

[15] Donko Donjerkovic, Yannis E. Ioannidis, Raghu Ramakrishnan: Dynamic Histograms: Capturing Evolving Data Sets. ICDE 2000: 86

[16] Phillip B. Gibbons, Yossi Matias, Viswanath Poosala: Fast Incremental Maintenance of Approximate Histograms. VLDB 1997: 466-475

[17] V. Nageshwara Rao, Vipin Kumar: Concurrent Access of Priority Queues. IEEE Transactions on Computers 37(12): 1657-1665 (1988)

[18] Jim Gray, Prashant J. Shenoy: Rules of Thumb in Data Engineering. ICDE 2000: 3-12

[19] Ibarra, O. H., and Kim, C. E. (1975), Fast approximation for the knapsack and sum of subset problems, *J. ACM* **22**, 463-468.

[20] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, editor, *Surveys in Combinatorial Optimization*, volume 31 of *Annals of Discrete Mathematics*, 1987: 213-258.

Appendix

In the following we will show that the approximation achieved by the greedy heuristic in our setting is close to optimal. Remember that we use the same greedy algorithm as the Knapsack problem in order to decide which results to keep in the cache and which to evict.

Assume that B_{opt} is the benefit of the optimal algorithm, and B_{gr} the benefit achieved by the greedy heuristic. Then, we wish to show that the ratio $r = (B_{opt} - B_{gr})/B_{opt}$ is small, or in other words, that the solution achieved by greedy is very close to optimal. We know that for the fractional Knapsack problem the optimal solution is the one given by greedy $B_{opt} = B_{gr} + f{\cdot}b_{last}$, where $0 \leq f < 1$, and b_{last} is the benefit of the last item, a fraction of which was taken. Then, it holds that:

$$r = \frac{B_{opt} - B_{gr}}{B_{opt}} \leq \frac{f \cdot b_{last}}{B_{opt}} \leq \frac{f \cdot b_{last}}{B_{gr}},$$

and

$$B_{gr} \geq \frac{S}{s_{\max}} b_{\min},$$

where S is the total size of the cache, s_{max} is the size of the largest item, and b_{min} the smallest benefit. By combining the above two formulas we get for r:

$$r \leq \frac{f \cdot b_{last}}{\frac{S}{s_{\max}} b_{\min}} \leq \frac{f \cdot b_{\max}}{\frac{S}{s_{\max}} b_{\min}},$$

where b_{max} is the largest benefit. In the last inequality we observe that if $s_{max} << S$ and b_{max} is close to b_{min} then the ratio r is very close to 1. In a commercial database management system we expect to have several hundreds of cached results of relatively small size. Therefore, the first assumption is almost certain to hold. We could argue that the same is true for the second assumption as well, given that there are no extreme cases in the workload under consideration. Consequently, the statement that the greedy approximation is close to optimal for our setting is true in the general case.