Improved Techniques for Result Caching in Web Search Engines

Qingqing Gan
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY 11201, USA
qq_gan@cis.poly.edu

Torsten Suel*
Yahoo! Research
701 1st Avenue
Sunnyvale, CA 94089
suel@poly.edu

ABSTRACT

Query processing is a major cost factor in operating large web search engines. In this paper, we study query result caching, one of the main techniques used to optimize query processing performance. Our first contribution is a study of result caching as a weighted caching problem. Most previous work has focused on optimizing cache hit ratios, but given that processing costs of queries can vary very significantly we argue that total cost savings also need to be considered. We describe and evaluate several algorithms for weighted result caching, and study the impact of Zipf-based query distributions on result caching. Our second and main contribution is a new set of feature-based cache eviction policies that achieve significant improvements over all previous methods, substantially narrowing the existing performance gap to the theoretically optimal (clairvoyant) method. Finally, using the same approach, we also obtain performance gains for the related problem of inverted list caching.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval.

General Terms

Algorithms, Performance

Keywords

Search Engines, Result Caching, Index Caching, Weighted Caching

1. INTRODUCTION

Large web search engines need to be able to process thousands of queries per second on collections of billions of web pages. As a result, query processing is a major performance bottleneck and cost factor in current search engines, and a number of techniques are employed to increase query throughput, including massively parallel processing, index compression, early termination, and caching. In particular, each query is routed to a large number of machines, say a few hundred or thousand, that process it in parallel. Index compression is used to decrease the sizes of the index structures, thus significantly reducing data transfers between disks, main memory, and CPU. Various early termination techniques are used to identify the best results without traversing the full index structure of each term in the query. Finally, caching is employed at several levels to reduce system load.

In this paper, we focus on the last technique, caching, and in particular the caching of query results. Caching is a common performance optimization technique in many scenarios including operating systems, databases, and web servers. In the case of web search engines, caching happens at several levels. At the lower level of the system, index structures of frequently used query terms are cached in main memory to save on disk transfers [16, 3, 1]. At the higher level, results of frequently asked queries are cached so that results can be returned without executing the same queries over and over again [15, 18, 12, 3, 7, 1, 2]. In addition, there may be other intermediate levels of caching that store partially computed results, say for common combinations of words [13].

Our contribution in this paper can be split into two parts, focusing on weighted caching and on feature-based caching. In the first part, we consider result caching as a weighted caching problem. Previous work has looked at result caching as an unweighted problem, where each query has the same cost and the goal is to maximize the number of queries that are served directly from cache. However, real query processing costs in search engines vary significantly among queries. For example, queries with very common terms or with many terms can be much more expensive than other queries. Thus, while optimizing hit ratio may be useful to minimize end user latency, it does not maximize query throughput. Thus, to minimize overall query processing costs, a good caching mechanism needs to also take the actual costs of the queries into account.

We discuss and evaluate several weighted caching algorithms for result caching, including weighted counterparts of LFU, LRU, and the state-of-the-art (unweighted) SDC policy in [7], as well as several new hybrid algorithms. In this context, we also address an interesting problem arising in connection with Zipf-based query distributions, or in general with distributions where large numbers of (potential) queries occur very rarely or not at all. In a nutshell, in such cases naive approaches may overestimate the frequencies of rare queries that just happened to occur by chance. As a result, we may end up caching too many items with high potential benefit (cost savings) that never occur again. We discuss several possible ways to address this problem.

Our second contribution is a new set of feature-based cache eviction policies that achieve significant improvements in hit ratio, and moderate improvements in cost, over existing methods. In particular, these policies significantly narrow the gap between the best known algorithm and the upper bound given by the clairvoyant algorithm. We also show that the same approach can be used to obtain improvements over current list caching policies.

These new policies are based on two observations. First,

*Current Affiliation: CSE Dept., Polytechnic Inst. of NYU

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

query traces have a lot of interesting application-specific structure that is not exploited by existing cache eviction policies that only rely on past occurrences of the same query. We can expose some of this structure to the caching mechanism via new features such as query lengths or the frequencies of individual query terms in the query log or collection. We note that a similar approach was also recently proposed in [2] in the context of a cache admission policy, and that researchers, e.g., in computer architecture have used application specific features for similar tasks such as branch prediction and data prefetching. We show here that in the case of eviction policies for search engine result caching, adding such features can make a significant difference in cache performance. Second, instead of making explicit use of such features inside new specially designed algorithms, it is probably smarter to rely on general statistical or machine learning techniques to make eviction decisions. This is particularly the case given the very large amount of query log data available to current search engines, and given that the cost of even fairly complicated eviction policies is small compared to that of executing a query. In our case, we utilize a very naive statistical approach that nonetheless gives significant benefits.

The remainder of this paper is organized as follows. Next, we provide some background on search engine architecture and discuss relevant previous work. In Section 3 we describe our experimental setup. Section 4 contains our results for weighted caching. Section 5 presents and evaluates feature-based eviction policies, and finally Section 6 provides concluding remarks.

2. BACKGROUND AND PREVIOUS WORK

Background on Search Engines: The basic functions of a crawl-based web search engine can be divided into four stages: data acquisition (or crawling), data mining and preprocessing, index construction, and query processing. During crawling, pages are fetched from the web at high speed, either continuously or through a set of discrete crawls. Then various data mining and preprocessing operations are performed on the data, e.g., detection of web spam or duplicates, or link analysis based on PageRank [4]. Third, a text index structure is built on the preprocessed data to support fast query processing. Finally, when a user issues a query, the top results for the query are retrieved by accessing the index structure. We focus on this last stage.

Current search engines are based on an index structure called an *inverted index* that allows us to efficiently identify documents that contain a particular term or set of terms [21]. To do so, an inverted index contains an *inverted list* for each distinct term w that occurs somewhere in the collection; this is basically a list of the IDs of those pages in the collection that contain w, often together with other data such as the number of occurrences in the page and their locations. A query is processed by fetching and traversing the inverted lists of the search terms, and then ranking the encountered pages according to relevance. Note that the length of an inverted list increases with the size of the collection, and can easily reach hundreds of MB or several GB even in highly compressed form. Thus, for each query a significant amount of data may have to be fetched and processed. This is the main performance challenge in search engine query processing, and it has motivated various performance optimizations.

Caching in Search Engines: One such optimization is the use of caching, which occurs in search engines on two levels. A query enters the search engine via a *query integrator* node that is in charge of forwarding it to a number of machines and then combining the results returned by those machines. Before this is done, however, a lookup is performed into a cache of previously issued queries and their results. Thus, if the same query has been recently issued, by the same or another user, then we do not have to recompute the entire query but can simply return the cached result. This approach, called *result caching*, is widely used in current engines, and has also been studied by a number of researchers [15, 18, 12, 3, 7, 1, 2]. A second form of caching, called *index caching* or *list caching*, is used on a lower level in each participating machine to keep the inverted lists of frequently used search terms in main memory [9, 16, 3, 20].

Our main focus is on result caching. This approach has been shown to achieve significant performance benefits on typical search engine traces. Note that the performance depends on the characteristics of the queries posed by the users, the policies used by the search engine to process queries (e.g., whether the order of terms in the query matter, or whether users from different locations receive different results), and the caching policies that are used. Our goal is to design caching policies that best exploit the properties of typical search engine query logs to achieve a high cache hit ratio and large cost savings. Earlier work on search engine query logs and result caching (see, e.g., [17, 15, 18, 3] has shown that such logs have several interesting properties:

- Query frequencies follow a Zipf distribution.
- While a few queries are quite frequent, a significant fraction of all queries occur only once or a few times.
- Query traces exhibit some amount of burstiness, i.e., occurrences of queries are often clustered in a few time intervals.
- A significant part of this burstiness is due to the same user reissuing a query to the engine.

Previous Work on Result Caching: The first published work on result caching in search engines appears to be the work of Markatos in [15], which studies query log distributions and compares several basic caching algorithms. Work in [18] looks at various forms of locality in query logs and proposes to cache results closer to the user. Work by Lempel and Moran [12] proposes improved caching schemes for dealing with requests for additional result pages (i.e., when a user requests a second or third page of results). Several authors [16, 3, 1, 8] have considered the impact of combining result caching and list caching; in particular, recent work in [1] studies how to best share a limited amount of memory between these two forms of caching. In [8], Garcia examines caches for the query evaluation process as a whole. Finally, work in [7, 2] considers hybrid methods for result caching that combine a dynamic cache that exploits bursty queries with a more static cache for queries that stay popular over a longer period of time. We discuss related work in more detail later as necessary in the context of our own results.

Weighted Caching: Previous work on result caching [15, 18, 12, 3, 7, 1, 2] has focused on maximizing the hit ratio of the cache, that is, the percentage of queries that can be answered directly from cache. We note here that caching has two possible objectives: (a) reducing the delay experienced by the user by keeping cached results closer to the user (either in terms of network distance or memory hierarchy), and (b) reducing the load on the underlying system by avoiding unnecessary computations (which may in turn also reduce

delays seen by the user). Maximizing the hit ratio focuses on the delay, which is an important objective particularly for forward result caches that are deployed outside the main search engine cluster and closer to the user.

However, we argue that the second objective is also very important in current search engines, which have invested hundreds of millions of dollars in hardware for query processing. We note that search queries vary dramatically in terms of their computational cost depending, e.g., on the number of search terms and their frequency in the underlying collection and in other search queries. (The latter determines how likely the corresponding inverted lists are to already be in main memory.) The exact costs for a query of course depend on the internal design of the engine. But it seems realistic to assume that when a query is computed, some measure of the overall computational cost is communicated to the result caching mechanism, which should then use this cost in its eviction policy to give preference to retaining results of expensive queries.

We consider both objectives in this paper. Each query result has the same size (maybe a few KB for storing the result URLs and result snippets), but queries can have very different benefits (cost savings when reused) associated with them, and our goal is to maximize the total benefit that is achieved. This is an instance of a *weighted caching problem* studied, e.g., in [5, 19]. We note that standard algorithms for caching such as LRU (Least Recently Used) or LFU (Least Frequently Used) do not take benefits into account and thus do not perform well on such problems. The work in [5, 19] proposes a caching algorithm called *Landlord* that is essentially a generalization of LRU; this algorithm assigns leases to items based on their sizes and benefits and evicts the object with the earliest expiring lease. The Landlord algorithm was recently adapted to another caching problem in search engines in [13].

Cache Admission Policies: Researchers in several areas have recently proposed cache admission policies that prevent certain items from even being inserted into the cache [14, 13, 2]. In the context of web search engines, cache admission was used in [13] in a scenario where cache insertion itself has a significant cost that would be wasted if the item is evicted soon afterwards without having been reused. In [2] a cache admission policy is used to predict and eliminate query results that are unlikely to occur again. We note that in this latter case, cache admission is really used as a mechanism for improving a non-optimal eviction policy (since an insertion followed by immediate eviction from a cache with one additional slot would have the same effect, as opposed to [13] where insertion incurs a significant cost). The work in [2] is the previous work most closely related to our feature-based eviction policy in that it also suggests the use of application-specific features. One difference to our work is that [2] uses these features to make a 0/1 decision about admission of an item, while we use them to predict a probability of reoccurrence that is used by the cache eviction policy. We also consider a wider range of features in our experiments.

3. DATA AND EXPERIMENTAL SETUP

Given that search query logs have a very long tail of queries that occur only once or twice, it is important to use a fairly large query log to evaluate result caching policies. For our experiments, we used a log of 36,389,567 queries submitted to the AOL Search service between March 1 and May 31, 2006. We preprocessed the query log by removing stopwords, and completely removed any queries consisting only of stopwords. For all the results presented in this submission, we also removed requests for further result pages; such requests for the second, third, etc., page of results are best handled with other techniques as shown in [12], and most previous work does not spell out if such requests were retained or not. We assume that term order in queries is important, and that two queries are identical only if they contain the same words in the same order. However, for these last two choices, we also ran experiments that make the opposite choice; the results were overall fairly similar (in terms of the relative ordering of the methods) and are omitted from this paper.

The resulting query trace had 17,448,985 queries, including 10,087,344 distinct queries. Of these queries, 5,605,830 occurred only once, and 1,005,241 occurred exactly twice. In Figure 1 we plot the query frequencies using a double-logarithmic scale, where queries are ordered along the x-axis from most frequent to least frequent, and the frequency of the query is shown on the y-axis. We obtain a slope corresponding to $z = 0.82$, which compares to numbers between $z = 0.59$ [16] and $z = 0.86$ in the literature.

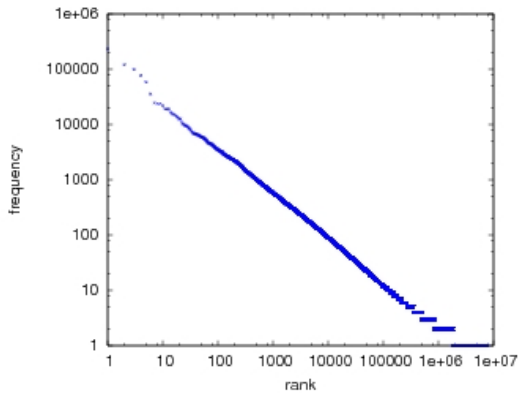

Figure 1: Query frequency distribution of the AOL data set.

In our experiments, we assume that every query has a computational cost that is taken into account by the weighted caching policies. This cost of course depend on details of each particular search engine's query processing system, which are considered proprietary. We experimented with several possible cost functions that define the cost of a query as follows: (i) the sum of the lengths of the inverted lists of the query terms, $\sum_{i=0}^{k} L_i$, (ii) the length of the shortest list L_0, and (iii) the product of L_0 and $\log_2(L_1/L_0)$ where L_1 is the second shortest list. These choices represent different bottlenecks in query execution: (i) when the cost of fetching lists from disk is dominant and the entire list has to be fetched, and (ii) and (iii) when index data is mostly in main memory and cost is dominated by the CPU cost of traversing the shortest list and looking up its elements in the next larger list. We obtained the lengths of inverted lists by running all queries against a subset of 7.5 million pages taken at random from a large web crawl.

We note that real cost functions will be more complicated and may have to take into account the impact of techniques such as index caching and early termination at the lower level. However, the caching algorithm does not need to know the function, as long as some cost estimate is returned by the query processor upon computing a query. We tried all

three cost functions and did not observe major changes in the relative performance of our algorithms. All results presented in this paper use cost function (i) above.

4. WEIGHTED RESULT CACHING

In this section, we investigate algorithms for weighted result caching. As mentioned before, weighted caching algorithms assume that different objects have different weights (in our case, costs of recomputing a query) that need to be taken into account in the eviction policy. We will consider both hit ratio and cost savings in the evaluation of our algorithms. We first introduce a few unweighted baseline algorithms, followed by their weighted counterparts. After that we introduce some improvements to these baseline algorithms by integrating additional ideas. In particular, we design hybrid algorithms that take advantage of burstiness in the query stream, and we study caching under Zipfian query distributions.

4.1 Baseline Methods

Before explaining our baseline caching methods, we describe two offline algorithms that can be used as upper bounds on the performance of the online algorithms. Our offline algorithm for hit ratio is the well-known clairvoyant algorithm that always evicts the object whose next access is farthest in the future, and it is known to achieve the best possible hit ratio. For the weighted case, however, we are not aware of any polynomial-time offline algorithm that guarantees the highest possible cost savings. Instead, we design a heuristic, called "Future_Known", that we hope will approach the optimal solution in most cases. In this algorithm we compute the priority score of a query by dividing the cost of a query by the distance from the current to the next occurrence of this query. Then, at each step, the query with minimum score is evicted from the cache.

The following eviction policies have been extensively studied in the context of result caching:

- **LRU (Least Recently Used):** When the cache is full, we always evict the least recently seen query. LRU is one of the most common cache eviction policies in computer science.
- **LFU (Least Frequently Used):** LFU evicts the least popular query. In our implementation, in addition to the frequency scores stored in the cache, we also keep some limited history of frequencies for queries that have been evicted from cache. In practice, setting this history to two or three times the number of items in cache achieves most of the available benefit.
- **SDC (Static and Dynamic caching):** As a hybrid algorithms, SDC [7] splits the cache into two parts, with one containing a static set of results for the most frequent queries, and the other using LRU for dynamic caching. In practice, we use 20% of the total cache size for LRU, which gave the best results.

The above methods are geared towards optimizing the hit ratio, i.e., the percentage of queries that can be served from cache. However, given the large variations in query processing costs between different queries, a more meaningful measurement should look at the hit ratio as well as the cost of processing a particular query. We can model this cost by a weighted caching problem. The following are natural weighted counterparts of the above unweighted algorithms that are frequently studied:

- **Landlord:** In the Landlord algorithm [5, 19], whenever an object is inserted into the cache, it is assigned a deadline given by its cost. We always evict the element with the smallest deadline, and deduct this deadline from the deadlines of all other elements currently in the cache. (Instead of actually deducting from all entries, the algorithm is best implemented by summing up all values of deadlines that should have been deducted so far, and taking this sum properly into account.)
- **LFU_w:** The only difference to LFU is that the priority score is now the product of frequency and cost.
- **SDC_w:** This method is motivated by SDC. The difference is that we use LFU_w for the static part and Landlord for the dynamic part of the cache.

We studied the performance of the above baseline algorithms using cache sizes ranging from $25k$ to $800k$ items. In Figure 2, we compare the algorithms by measuring their hit ratios. In addition, we also plot the result of the Clairvoyant algorithm for comparison. As expected, for the unweighted versions, SDC performs better than LRU, and they both win over LFU. The ordering is the same for the weighted versions of these three algorithms, but of course these algorithms perform worse than their unweighted versions when looking at hit ratio. In Figure 3, we plot the results when the same algorithms are evaluated according to cost savings. As expected, the weighted algorithms now perform better than the unweighted ones, but the order within versions is still the same. Overall, SDC is the best algorithm for hit ratio, and SDC_w the best algorithm for cost savings.

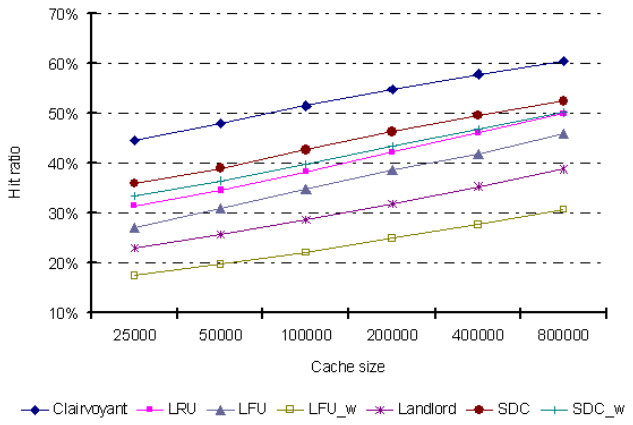

Figure 2: Performance of baseline caching algorithms (hit ratio). From top to bottom, the seven lines shown are for Clairvoyant, SDC, SDC_w, LRU, LFU, Landlord, and LFU_w.

4.2 Generalized Hybrid Algorithms

As mentioned, query logs are known to be very bursty [17]. Thus, queries are more likely to reoccur shortly after another occurrence. The sources of query burstiness could be due to two reasons: The same query is repeatedly issued by the same user, or there is a burst of global popularity for a particular query. In this section, we first look at the

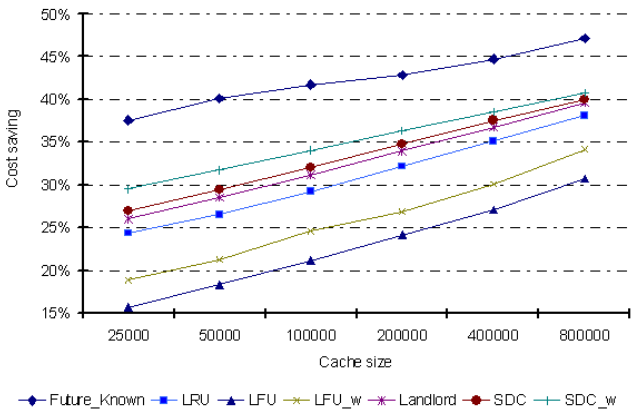

Figure 3: Performance of baseline caching algorithms (cost savings). From top to bottom, the seven lines shown are for Future_Known, SDC_w, SDC, Landlord, LRU, LFU_w, and LFU.

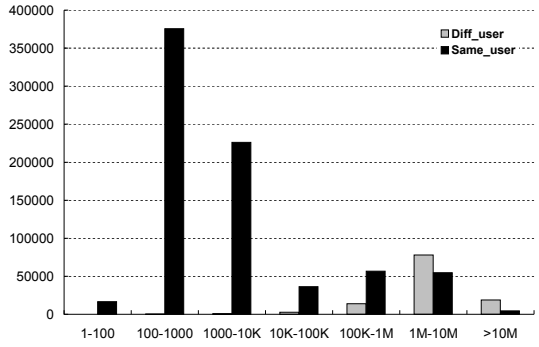

Figure 4: Burstiness in the AOL query log for queries occurring exactly twice.

amount of burstiness in the AOL trace, and then propose hybrid algorithms that exploit it.

If the cache size is small, only queries that have occurred a large number of times will be placed into a static LFU cache before the testing queries arrive. For example, for a cache size of $10k$ items, only queries that have occurred more than 81 times will be in cache. Even for a cache size of $100k$ items, only queries that occur at least 19 times are in cache. However, queries that occur no more than twice account for 43.4% of all queries issued. Thus, queries that occur very few times are of particular concern as they are not likely to be in a static cache but make up a large portion of the total query load. Consequently, any caching algorithm focusing only on frequency is likely to not do very well.

Clearly, queries that occur only once cannot be cached, but queries occurring 2, 3, or a small number of times could be amenable to caching even with very limited cache sizes if their occurrences are very bursty. To check this, we now consider those queries that occur exactly twice. In Figure 4, the x-axis shows the number of other queries between the first and second occurrence of the same query, while the y-axis shows the number of queries falling into each class on the x-axis. For each class, we show two bars, one for queries issued twice by the same user, and one for queries issued by two different users. We observe that there is significant burstiness, and that most of it arises from the same user reissuing a query. In particular, assuming an arrival rate of 132 queries per minute (the average rate over the trace), we can see that most queries are reissued within a few minutes to at most an hour. Also, almost all queries reissued within about a day in our query log are by the same user.

Thus, if we could add a small cache to temporarily hold query results for at least a few minutes, this would allow us to capture many repeats of queries that occur only a few times. Actually, this was one of the main motivations for the SDC algorithm in [7]. In the following, we present another variation of this idea that results in additional benefits over SDC and its weighted variant from the previous subsection.

The basic idea underlying hybrid algorithms can be described as follows. The cache space is partitioned and each partition is administered by a different caching policy. For example, in SDC, a small fraction of the space, usually 10% to 30%, is ruled according to LRU, while the rest is used to statically store the most frequent queries. In this way, the burstiness is captured by the LRU part of the cache. The results in [7] showed that SDC outperforms other pure (single policy) caching algorithms in terms of hit ratio. (Note that a hybrid algorithm could also dynamically change the division of space between policies, but we did not find any benefit in this for result caching.)

We now further generalize SDC for the weighted case as follows: Assume two caching policies A and B are available, and that the total space is divided between A and B in a fixed way. Each policy assigns a score to each query (e.g., time since last occurrence for LRU); this means that each cached query is associated with two scores, one for each policy. After executing a query, we attempt to insert it into either cache. When we evict an item from one cache, say the cache associated with A, we select the lowest scoring item according to policy A. However, before throwing the item out, we first attempt to insert it into the other cache. If the item has a high enough score according to policy B, this will result in the eviction of another item from cache B. In principle, this could continue for a number of steps, but eventually an item is evicted, and this usually happens after at most a few steps.

We now evaluate the performance of two hybrid approaches, one combining LRU with LFU_w (Hybrid1), and one combining Landlord and LFU_w (Hybrid2), under this framework. We found that using about 80% of the space for LFU_w gave the best results overall. In Figure 5, we show the results for cost saving. We observe visible improvements over SDC_w, the weighted version of SDC from the previous subsection, with Hybrid2 obtaining the best performance. We note here that the two new hybrids are more dynamic in that queries can enter the (usually fairly static) LFU_w cache after spending some time in the other cache, and that checking items against both policies appears to have benefits over running two completely independent caches.

4.3 Estimation Corrector

Recall the setup of the result caching problem: Given our observation of past query occurrences, how do we de-

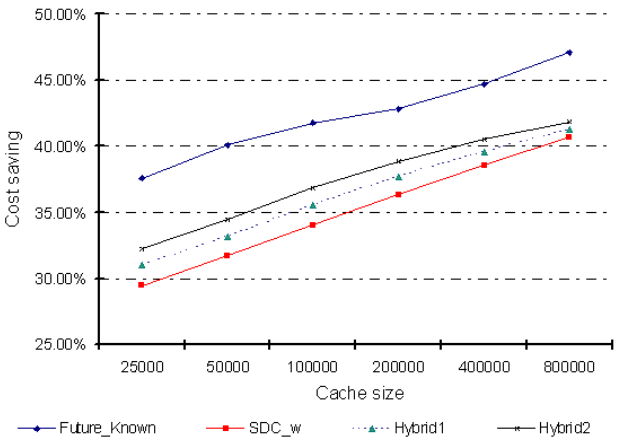

Figure 5: Cost reduction for hybrid algorithms. From top to bottom, the four lines shown are for Future_Known, Hybrid2 (Landlord+LFU_w), Hybrid1 (LRU+LFU_w), and SDC_w.

cide which query result to remove from cache? To make this decision, we need a good estimate of the probability of reoccurrence of a query x, called $Pr(x)$. Assuming stable probabilities, a simple estimate is to take the current value of the frequency of a particular query divided by the total number of queries seen so far. In fact, this is the underlying idea of LFU. As we will argue, for certain cases, this is not a good estimate. In the following we discuss the problem of estimating probabilities of query reoccurrences given a typical highly skewed query distribution.

As shown in Section 3, the query log is following a Zipf-based distribution where a small portion of queries have a high frequency but most queries have a fairly small probability of occurring. In such a case the frequency of past occurrences by itself does not provide the best estimate for future occurrences. In particular, one occurrence of a query may mean much less than two occurrences. For example, suppose lottery tickets are sold according to a Zipfian distribution such that a few people buy a lot of tickets but many people buy only one or two tickets. Each player buys the same number of tickets each week, and each week there is one winner. We observe the winners of the lottery over many weeks, and have to guess how many tickets they bought (which is proportional to the probability of them winning in future rounds). People who buy few tickets may win once (because say 50% of tickets are sold to people who buy only one ticket) but will very rarely win twice. So if we observe the same person winning twice, this would indicate they likely bought more than twice as many tickets as a person winning once. To apply a similar idea to our case, if a query occurs only once, it is often a very rare query from the long tail of the query distribution. It could be due to a typo by a user, or due to a combination of rare query terms. When a query occurs twice, on the other hand, its chance of reoccurrence might be more than twice as high. This idea is of course closely related to smoothing techniques in language modeling, see, e.g., [6].

We consider three different ways to get a more precise estimate of the probability of a query reoccurring in the future given its past occurrences: (1) an approach based on the Good-Turing method, (2) an approach based on a formal analysis of Zipfian distributions, and (3) a heuristic approach that simply applies a set of weights derived from an actual query trace (but which subsumes the other approaches).

Our first method is motivated by the Good-Turing estimator [6], often used in the context of smoothing techniques for language modeling, but applied here to result caching. The basic Good-Turing estimator for the likelihood of an occurrence in the next step can be stated as:

$$Pr_x = \frac{N_x + 1}{T} \cdot \frac{E(N_x + 1)}{E(N_x)},$$

where, in our case, x is the query and N_x is the number of times query x has occurred so far. T is the number of queries observed thusfar, and $E(n)$ is the number of different queries that have occurred exactly n times.

Our second method is based on the analysis of a Zipfian probability distribution to estimate the likelihood of a query reoccurring. We assume a sequence of events $\{u_1, ..., u_d\}$ that is generated following a Zipfian distribution. Given that query x has occurred k times in the past, Pr_x can be estimated as:

$$Pr_x = \sum_{i=1}^{d} \left(\frac{p_i \cdot B(p_i, h, k)}{\sum_{j=1}^{d} B(p_j, h, k)} \right), \quad (1)$$

where p_i is the probability of choosing u_i, and $B(p_i, h, k) = \binom{h}{k} p_i^k \cdot (1 - p_i)^{h-k}$. For more details, see the Appendix.

An alternative to the above two approaches is to derive a penalty function $g(f_x)$ experimentally from a query log used as a training set. This penalty function is then multiplied with the naive estimate (number of past occurrences over the number of observed queries) to get a better estimate. In fact, for larger values of f_x, say $f_x > 20$, $g()$ is very close to 1.0 since the naive estimate is quite precise. Thus, we only need to determine a small table of penalty values for at most 20 values of f_x. To do this, we performed a simple search procedure using a query trace to derive the optimal setting of the values $g(f_x)$, using the formally derived numbers from the second method as starting points. In fact, we found this approach to be more flexible in practice, as real query logs are not accurately modeled by a clean mathematical distribution. It also slightly outperformed the other two methods and is thus used in our experiments.

We note that the discussion in this subsection is only relevant for the case of weighted caching. For unweighted caching problems, adding any of the above correction mechanisms would have no impact on cache behavior, as long as the probability of a reoccurrence is monotonically increasing with the number of past ccurrences. However, this is different for weighted caching problems, e.g., when deciding whether to cache a frequently seen query with low cost, or a rarely seen query with high cost.

In Figure 6, we show results comparing the best of the previous algorithms to versions that apply experimentally determined penalty factors $g()$. Overall, the new versions outperform the previous best versions by a moderate but visible amount.

5. FEATURE-BASED CACHING

In this section, we introduce feature-based cache eviction policies for result caching, and also for the related problem of list caching. The basic idea is that search engine query logs contain a lot of features beyond the previous occurrences of

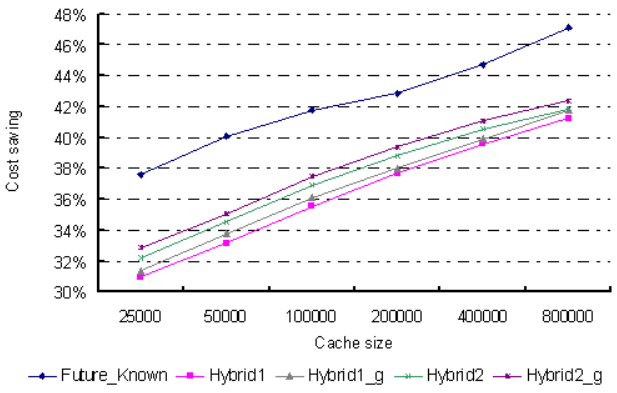

Figure 6: Cost reduction of the caching algorithms by adding a penalty factor g. From top to bottom, the five lines shown are for Future_Known, Hybrid2_g, Hybrid2, Hybrid1_g, and Hybrid1.

a query that can be used to make caching decisions, and that such features are best incorporated into eviction policies via data analysis and machine-learning techniques rather than the design of explicit algorithms. This section consists of three parts. First, we discuss the basic idea and define ten features that we considered for result caching. Then we describe the approach that we used to derive eviction policies from these features. Finally, Subsection 5.3.1 presents experimental results for result caching, and Subsection 5.3.2 presents results for list caching.

5.1 Features

In much of the caching literature, including previous work on result caching in search engines, a workload is modeled as a sequence of integers where each integer identifies an object that is accessed in a given step. This gives a simple framework in which to design and analyze caching algorithms such as LRU or LFU, but it also obscures many application-dependent properties of the workload that might lead to better caching decisions. For example, simply representing each query in a search query log as an ID means that we can look at previous occurrences of exactly the same query to make caching decisions, but we cannot look at other potentially relevant properties such as occurrences of other queries that are similar to this query (e.g., queries that are a superset of our query), whether the query contains a very rare term, or even just the number of terms in our query. However, it is known that single-term queries are more likely to occur again than longer queries [15], and thus to optimize hit ratio it might be a good idea to give preference to shorter queries in caching decisions. In general, search engine query logs have a lot of interesting structure that can be mined for various purposes such as improved ranking or ad placement, and it seems reasonable to assume that this structure could also be exploited for better caching decisions.

To prove this conjecture, we first have to define a set of suitable features in the query trace that is likely to be useful in caching decisions. The set should certainly contain standard features such as the number of times a query has previously occurred and the time since the last occurrence, but also new features not used by traditional caching algorithms. After some exploration, we focused on the following ten features F1 to F10 that are considered for each query:

- F1: the number of steps since the last occurrence of this query. This feature is the basis for LRU.
- F2: the number of steps between the last two occurrences of this query, if a query has happened at least twice. Otherwise, F2 is set to *unavailable*. (Similarly, we could also add the number of steps between the second- and third-last occurrences as another feature.)
- F3: the query frequency up to this point. This is the feature underlying LFU.
- F4: the query length, defined as the number of terms in the query.
- F5: the length of the shortest inverted index list of any term in the query. This tests whether the query has a term that is rare in the collection.
- F6: the frequency of the rarest query term in the log.
- F7: the number of distinct users who issue this query. If user IDs are not available, IP addresses can be used instead.
- F8: the gap between the last two occurrences of the query that were issued by the most recently active user. As we saw in Section 4, a large portion of those queries that occur only two or three times are issued by the same user, and those queries are usually very close to each other in the query log.
- F9: the average number of clicks for the query. Intuitively, queries with more clicks might have a higher probability of occurring again. This is one but not the only way to harvest information from user clicks. We could also model the distribution of clicks in more detail, or use clicks to guess whether a query is navigational, using the approach in [11].
- F10: the frequency of the rarest pair of query terms in the query log. This measures if the query contains a pair that is only very rarely used together in queries.

It is easy to think about many additional features, but we found the above selection to be most promising. The features can be grouped into two categories: Traditional features (F1 to F3) that are used by many well-known caching algorithms, and non-traditional features (F4 to F10) that are specific to our application domain and that would be obscured if we only treat queries as objects with a unique ID.

In order to understand the usefulness of these features, we first looked at their information gain, as commonly studied in machine learning. Essentially, we want to check how useful each feature is in predicting reoccurrence of a query within a limited number of steps. The details of this experiment are a little tricky (and omitted due to space constraints), as we first had to determine appropriate threshold values to get a binary classification for each target cache size.

Features	F1	F2	F3	F4	F5
Cache Size=100k	0.247	0.029	0.114	0.017	0.001
Cache Size=200k	0.213	0.018	0.102	0.009	0.011
Cache Size=400k	0.287	0.009	0.098	0.008	0.029
Features	F6	F7	F8	F9	F10
Cache Size=100k	0.093	0.192	0.023	0.001	0.076
Cache Size=200k	0.056	0.101	0.003	0.012	0.089
Cache Size=400k	0.078	0.106	0.009	0.010	0.098

Table 1: Information Gain for Different Features and Cache Sizes.

The measured information gain scores for the features are shown in Table 1, for three different cache sizes. We can easily see that, in general, F1, F3, and F7 have the highest information gain, which means they should be very useful features when deciding which query should be evicted from cache. However, the remaining features also offer benefits. Not surprisingly, some of the features are more promising for smaller cache sizes, while other are best for larger ones.

5.2 Caching Mechanism

We now describe how to implement a cache eviction policy based on the above features. There are two aspects to this problem: How to use the features to predict the likelihood of a reoccurrence of a query, and how to efficiently identify the query with the smallest such likelihood.

For the first aspect, we started out with what may look like a very crude approach: We split the range of each of the features into a number of bins, in our case 8, so that each possible value falls into one of the bins. (For example, for F3, the number of previous occurrences of the query, we might have bins for $F3 = 1$, $F3 = 2$, $F3 = 3, 4$, $F3 = 5, 6, 7, 8$, etc.) Thus, at any point of time, any query in the cache belongs to one of 8^{10} buckets based on its current feature values. For each bucket, we also keep some historical statistics, in particular the total number of query instances that we have observed in this bucket, and how often these queries then reoccurred within a short period of time in the input sequence. Thus, for each bucket we essentially maintain a simple estimate of the likelihood of reoccurrence for queries that fall into the bucket; we then evict a query in the cache that has the lowest such likelihood (i.e., that belongs to the bucket with the lowest such likelihood).

A few more remarks about this approach, which may seem extremely naive from a machine-learning perspective. First, many of the buckets are empty and in fact we get by with far fewer than 8^{10} buckets, requiring only a few MB of total memory. (If space becomes a problem, we can use fewer buckets in areas with sparse data.) In contrast, each cached query result has a size of a few KB, and thus a cache for 100000 queries takes a few hundred MB. Second, we tried employing smarter approaches for exploiting the features, including interpolation between buckets and logistic regression on the feature values, but we observed at most very minor additional gains. Thus, we decided to stay with the simple bucket approach in our implementation. (In fact, this may be another example where data size beats algorithmic technique – for smaller query traces we would expect to see more improvements by using smarter techniques.)

Given this structure, we now have to implement an efficient mechanism for cache eviction. First, we note that any statistics needed in features F5, F6, and F10 are precomputed based on a sample of data; thus, e.g., the frequencies of different terms in the query trace used in F6 are precomputed from a sample of queries. In general, the features can be split into three classes: (1) Features such as F2, F3, F7, F8, and F9 that can only change their values when a query reoccurs; in that case, we can update the feature values of a query in the cache when it reoccurs and then move the query to its new bucket. (2) Features that are determined by the query itself based on a precomputation and that never change during the algorithm, e.g., F4, F5, F6, and F10. (3) Features, in our case only F1, that change as time proceeds. We can efficiently deal with F1 by maintaining a data structure on the time of last occurrence of a query, and moving queries to new buckets whenever they would have to move to a new bin according to F1. This means that apart from movements to new buckets that are triggered by reoccurrences of a query, a query can be moved at most 7 times to a new bucket along the F1 axis. Overall, this eviction mechanism can be implemented to run in $O(\log(n))$ steps per query in the workload using standard data structure, and is highly efficient in terms of actual processing overhead.

5.3 Experimental Results

In this section, we present experimental results for our feature-based caching approach, and compare it to existing algorithms. We first consider result caching in Section 5.3.1, and later extend the approach to list caching in Section 5.3.2. We first consider hit ratio, and then further below we look at cost savings in the weighted case. In all experiments, we used the first 10 million queries to initialize the statistics for each bucket. Then the following 5 million queries are used to warm up the cache, and the remaining 2.7 million queries are used to evaluate the actual caching performance. In general, of course, our feature-based approach benefits from the availability of sufficiently large query logs that can be used to derive statistics.

5.3.1 *Result caching*

In our first experiment, we compare a feature-based approach using only F1, F2, and F3 with existing algorithms, in particular SDC, LRU, LFU, and a variant of LFU that keeps a complete history of query occurrences. (This variant of LFU is used since basic LFU is at a disadvantage compared to SDC and our feature-based methods, which use larger tables of statistics). For the non-feature based algorithms, we used the first 15 million queries to initialize the cache, and then compared results on the remaining 2.7 million queries.

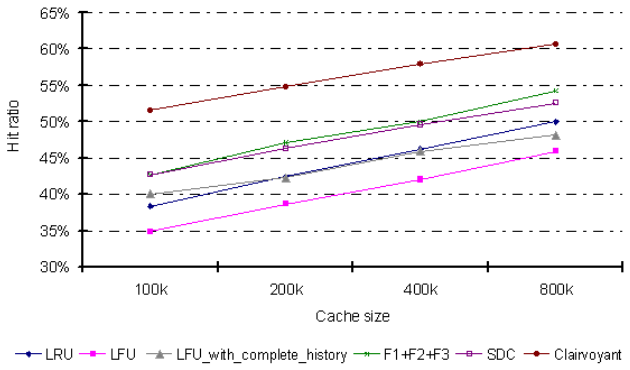

Figure 7: Hit ratios for feature-based caching with F1, F2, and F3, and for four existing algorithms. From top to bottom, the six lines shown are the hit ratios for the optimal clairvoyant algorithm, the feature-based method, SDC, LFU with complete history and LRU (overlapping), and basic LFU.

From the results in Figure 7 we see that the feature-based approach using only F1, F2, and F3 already slightly but consistently outperforms SDC, the best previous method for hit ratio. Note that F1, F2, and F3 are all traditional features, but there is no reason to believe that any of the previously known algorithms (such as SDC) exploits these features in

the best possible way. So even for this limited set of features, the results suggest that maybe learning from statistics is preferable to trying to design better explicit algorithms.

Next, we ran experiments with all ten features. The results are shown in Figure 8. We see that by using all ten features, we now get very substantial improvements over SDC, and are in fact able to significantly narrow the gap to the optimal off-line method. The best method on average outperforms SDC by about 4%, which is a fairly significant improvement given the amount of previous work on result caching.

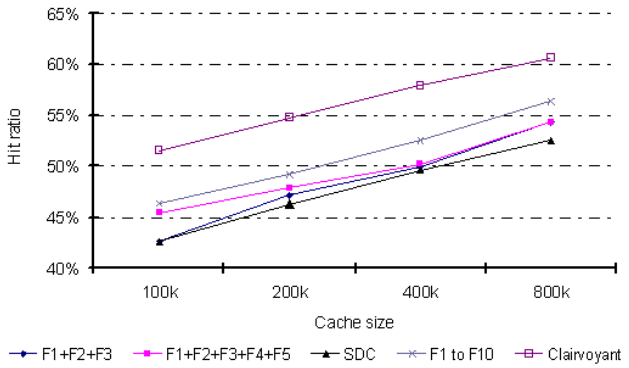

Figure 8: Hit ratios for feature-based caching with different combinations of features. From top to bottom, the five lines shown are the optimal clairvoyant algorithm, F1-F10, F1-F5, F1-F3, and SDC.

Next, we consider the weighted case, where the goal is to maximize the cost savings. To get a weighted algorithm using features, we simple multiply the cost of the query with our estimate of its likelihood of reoccurrence in the near future. As shown in Figure 9, our weighted feature-based method again outperforms all other methods, though by a smaller amount than in the case of hit ratio. Some improvements on this might be possible in future work, by introducing new features that are more relevant to the weighted case.

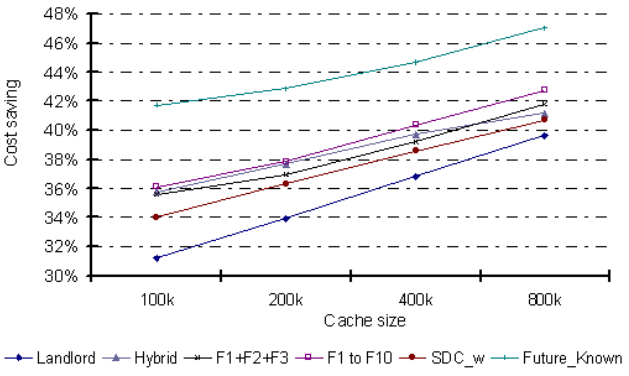

Figure 9: Results for weighted caching. From top to bottom, the six lines shown are the hit ratios for our best off-line algorithm, F1-F10, the best hybrid from Section 4.3 and F1-F3 (overlapping), SDC_w, and Landlord.

5.3.2 *List Caching*

Motivated by the success of our approach for result caching, we decided to also look at the related problem of list caching, i.e., the caching of inverted lists in main memory that is done at a lower level in the search engine. In previous work [1, 20], the best results for list caching were obtained by using either a fixed assignment of lists to the cache, or versions of LFU with additional memory. (In practice, the LFU version results in an almost static assignment, as lists tend to stay in the cache forever after being inserted.) However, there is still a significant performance gap between these methods and the upper bound given by the optimal off-line approach.

We note here that list caching is performed after doing result caching on the query log; this removes most of the short-term burstiness from the query stream. Also, we caution that a static or LFU method may not be appropriate for global search engines that observe a different mix of languages during different times of the day – in this case, a dynamic approach that changes the mix of cached lists during the day may be better. However, most publicly available traces are focused on a single search market. (The AOL trace used here is limited to the US market.) We defined the following features for each inverted list (and associated term) in the index:

- F1-F3: same as before, but defined on a per-term rather than per-query basis.
- F4: the number of distinct queries containing this term, divided by the total number of queries containing it.
- F5: the frequency of the most popular query containing this term.
- F6: the number of times this term has occurred as a single-term query.
- F7: the number of times this term has occurred as part of a two-term query.
- F8: the number of distinct users (or IP addresses) that have issued queries containing this term.

In Figure 10 we show the results for feature-based list caching. The objective in this case is to maximize the amount of data that is served from cache rather than disk. (Alternatively, we could also model disk cost savings more precisely by taking seek times into account; the results are very similar on our data.) We see that the feature-based approach outperforms all other methods by several percent, resulting in significant savings in disk traffic.

6. CONCLUDING REMARKS

In this paper, we have proposed and evaluated improved techniques for result caching in web search engines. In the first part of our work, we studied the weighted case, where our goal is to maximize cost savings instead of hit ratio. We described improved hybrid algorithms for this case that are particularly suitable for Zipf-based query distributions. In the second part of our work we proposed a feature-based approach to caching that achieves very significant improvements in hit ratios. Interestingly, the approach also provides improved results for the related problem of list caching.

Several interesting open problems remain. We plan to experiment with other features in query logs that might be helpful in predicting the likelihood of reoccurrence of a query, and that could lead to additional gains in hit ratio. A more formal analysis of burstiness in query logs, maybe starting

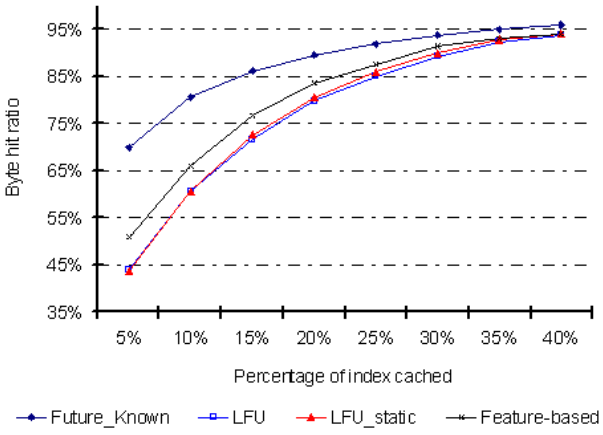

Figure 10: Byte hit ratios for the feature-based list caching algorithm. From top to bottom, the four lines shown are the hit ratios for our best off-line algorithm, F1-F8, and LFU and the static algorithm (overlapping).

from Kleinberg's model in [10], would also be of interest. Finally, future work on result caching should also look at the need for periodically refreshing cached results. It can be argued that in currently deployed search engines, hit ratios are limited much more by the need for fresh query results than by cache size constraints. To our knowledge, this issue has not been addressed by any published work.

Acknowledgements: We thanks Xiaojun Hei for collaboration in the early stages of this work, and Keith Ross and Dan Rubenstein for valuable discussions of caching under Zipfian distributions. This research was partly supported by NSF ITR Award CNS-0325777.

7. REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th Annual SIGIR Conf. on Research and Development in Information Retrieval*, July 2007.

[2] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. Witschel. Admission policies for caches of search engine results. In *Proc. of the 14th String Processing and Information Retrieval Symposium*, Sept. 2007.

[3] R. Baeza-Yates and F. Saint-Jean. A three-level search-engine index based in query log distribution. In *Proc. of the 10th String Processing and Information Retrieval Symposium*, Sept. 2003.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World Wide Web Conference*, 1998.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symp. on Internet Technologies and Systems (USITS)*, 1997.

[6] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling, 1996.

[7] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. on Information Systems*, 24, 2006.

[8] S. Garcia. *Search Engine Optimisation Using Past Queries.* Ph.D Thesis, Department of Computer Science and Information Technology, RMIT University, 2007.

[9] B. Jonsson, M. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 118–129, June 1998.

[10] J. Kleinberg. Bursty and hierarchical structure in streams. In *ACM SIGKDD*, pages 91–101, 2002.

[11] U. Lee, Z. Liu, and J. Cho. Automatic identification of user goals in Web search. In *Proc. of the 14th Int. World Wide Web Conference*, pages 391–400, 2005.

[12] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. of the 12th Int. World Wide Web Conference*, pages 19–28, 2003.

[13] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of the 14th Int. World Wide Web Conference*, May 2005.

[14] T. Malik, R. C. Burns, and A. Chaudhary. Bypass caching: making scientific databases good network citizens. In *Proc. of the 21st Int. Conf. on Data Engineering*, 2005.

[15] E. Markatos. On caching search engine query results. In *5th International Web Caching and Content Delivery Workshop*, May 2000.

[16] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 51–58, Sept. 2001.

[17] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a Very Large AltaVista Query Log. Technical Report 014, SRC (Digital, Palo Alto), Oct. 1998.

[18] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proc. IEEE Infocom*, 2002.

[19] N. Young. On-line file caching. In *Proc. of the 9th Annual ACM-SIAM Symp. on Discrete algorithms*, 1998.

[20] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. World Wide Web Conference*, April 2008.

[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.

APPENDIX
Zipf-based Probability Analysis

We now show our derivation of Equation (1) in Section 4.3. We assume a sequence of events $\{u_1, ..., u_d\}$ that is generated following a Zipfian distribution of probabilities. (Note that this is a simplification as this is not necessarily the same as the observed distribution.) Let p_i be the probability of choosing u_i, i.e., $p_i = i^{-z}/T$ where T is $\sum i^{-z}$ when $i = 1, 2..., d$. We assume that we know the size of the underlying universe U and the Zipf parameter z. Let X_k denote the fact that x has occurred k times. We estimate the probability that an item we have observed a certain number of times k will reoccur in the next step:

$$Pr[x \text{ occurs in the next step} \mid X_k] = \sum_{i=1}^{d} p_i \cdot Pr[x = u_i \mid X_k] \quad (2)$$

Using Bayes' Theorem, we get

$$Pr[x = u_i \mid X_k] = Pr[x{-}{>}k \mid x = u_i] \cdot \frac{Pr[x = u_i]}{Pr[X_k]} \quad (3)$$

If we pick an item x uniformly at random from all d possible items, we have $Pr[x = u_i] = 1/d$. Furthermore, using $B(p_i, h, k) = \binom{h}{k} p_i^k \cdot (1 - p_i)^{h-k}$ we get:

$$Pr[X_k] = \frac{1}{d} \cdot \sum_{i=1}^{d} B(p_i, h, k) \quad (4)$$

and

$$Pr[X_k \mid x = u_i] = B(p_i, h, k) \quad (5)$$

.

Putting this all together, we get

$$Pr[x \text{ occurs in the next step} \mid X_k] = \sum_{i=1}^{d} \left(\frac{p_i \cdot B(p_i, h, k)}{\sum_{j=1}^{d} B(p_j, h, k)} \right) \quad (6)$$

Thus, if we have observed that an item x has occurred k times in h steps, and knowing d and z but not taking into account any observations about how many times other items have occurred in these h steps, the above estimates the likelihood the item will occur again in the next step, or in any other step.