# The Impact of Caching on Search Engines

Ricardo Baeza-Yates[1]
rbaeza@acm.org

Aristides Gionis[1]
gionis@yahoo-inc.com

Flavio Junqueira[1]
fpj@yahoo-inc.com

Vanessa Murdock[1]
vmurdock@yahoo-inc.com

Vassilis Plachouras[1]
vassilis@yahoo-inc.com

Fabrizio Silvestri[2]
f.silvestri@isti.cnr.it

[1] Yahoo! Research Barcelona
Barcelona, SPAIN

[2] ISTI – CNR
Pisa, ITALY

## ABSTRACT

In this paper we study the trade-offs in designing efficient caching systems for Web search engines. We explore the impact of different approaches, such as static vs. dynamic caching, and caching query results vs. caching posting lists. Using a query log spanning a whole year we explore the limitations of caching and we demonstrate that caching posting lists can achieve higher hit rates than caching query answers. We propose a new algorithm for static caching of posting lists, which outperforms previous methods. We also study the problem of finding the optimal way to split the static cache between answers and posting lists. Finally, we measure how the changes in the query log affect the effectiveness of static caching, given our observation that the distribution of the queries changes slowly over time. Our results and observations are applicable to different levels of the data-access hierarchy, for instance, for a memory/disk layer or a broker/remote server layer.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – Search process; H.3.4 [**Information Storage and Retrieval**]: Systems and Software – Distributed systems, Performance evaluation (efficiency and effectiveness)

## General Terms

Algorithms, Experimentation

## Keywords

Caching, Web search, information retrieval systems, query logs

## 1. INTRODUCTION

Millions of queries are submitted daily to Web search engines, and users have high expectations of the quality and speed of the answers. As the searchable Web becomes larger and larger, with more than 20 billion pages to index, evaluating a single query requires processing large amounts of data. In such a setting, to achieve a fast response time and to increase the query throughput, using a cache is crucial.

The primary use of a cache memory is to speedup computation by exploiting frequently or recently used data, although reducing the workload to back-end servers is also a major goal. Caching can be applied at different levels with increasing response latencies or processing requirements. For example, the different levels may correspond to the main memory, the disk, or resources in a local or a wide area network.

The decision of what to cache is either off-line (static) or online (dynamic). A static cache is based on historical information and is periodically updated. A dynamic cache replaces entries according to the sequence of requests. When a new request arrives, the cache system decides whether to evict some entry from the cache in the case of a cache miss. Such online decisions are based on a cache policy, and several different policies have been studied in the past.

For a search engine, there are two possible ways to use a cache memory:

**Caching answers:** As the engine returns answers to a particular query, it may decide to store these answers to resolve future queries.

**Caching terms:** As the engine evaluates a particular query, it may decide to store in memory the posting lists of the involved query terms. Often the whole set of posting lists does not fit in memory, and consequently, the engine has to select a small set to keep in memory and speed up query processing.

Returning an answer to a query that already exists in the cache is more efficient than computing the answer using cached posting lists. On the other hand, previously unseen queries occur more often than previously unseen terms, implying a higher miss rate for cached answers.

Caching of posting lists has additional challenges. As posting lists have variable size, caching them dynamically is not very efficient, due to the complexity in terms of efficiency and space, and the skewed distribution of the query stream, as shown later. Static caching of posting lists poses even more challenges: when deciding which terms to cache one faces the trade-off between frequently queried terms and terms with small posting lists that are space efficient. Finally, before deciding to adopt a static caching policy the query stream should be analyzed to verify that its characteristics do not change rapidly over time.

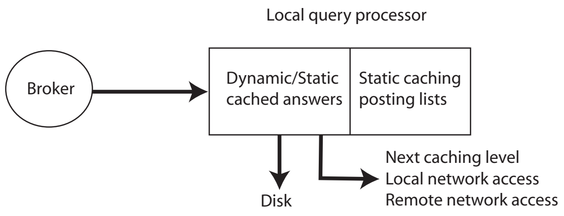

**Figure 1: One caching level in a distributed search architecture.**

In this paper we explore the trade-offs in the design of each cache level, showing that the problem is the same and only a few parameters change. In general, we assume that each level of caching in a distributed search architecture is similar to that shown in Figure 1. We use a query log spanning a whole year to explore the limitations of dynamically caching query answers or posting lists for query terms.

More concretely, our main conclusions are that:

- Caching query answers results in lower hit ratios compared to caching of posting lists for query terms, but it is faster because there is no need for query evaluation. We provide a framework for the analysis of the trade-off between static caching of query answers and posting lists;
- Static caching of terms can be more effective than dynamic caching with, for example, LRU. We provide algorithms based on the KNAPSACK problem for selecting the posting lists to put in a static cache, and we show improvements over previous work, achieving a hit ratio over 90%;
- Changes of the query distribution over time have little impact on static caching.

The remainder of this paper is organized as follows. Sections 2 and 3 summarize related work and characterize the data sets we use. Section 4 discusses the limitations of dynamic caching. Sections 5 and 6 introduce algorithms for caching posting lists, and a theoretical framework for the analysis of static caching, respectively. Section 7 discusses the impact of changes in the query distribution on static caching, and Section 8 provides concluding remarks.

## 2. RELATED WORK

There is a large body of work devoted to query optimization. Buckley and Lewit [3], in one of the earliest works, take a term-at-a-time approach to deciding when inverted lists need not be further examined. More recent examples demonstrate that the top $k$ documents for a query can be returned without the need for evaluating the complete set of posting lists [1, 4, 15]. Although these approaches seek to improve query processing efficiency, they differ from our current work in that they do not consider caching. They may be considered separate and complementary to a cache-based approach.

Raghavan and Sever [12], in one of the first papers on exploiting user query history, propose using a *query base*, built upon a set of persistent "optimal" queries submitted in the past, to improve the retrieval effectiveness for similar future queries. Markatos [10] shows the existence of temporal locality in queries, and compares the performance of different caching policies. Based on the observations of Markatos, Lempel and Moran propose a new caching policy, called Probabilistic Driven Caching, by attempting to estimate the probability distribution of all possible queries submitted to a search engine [8]. Fagni *et al.* follow Markatos' work by showing that combining static and dynamic caching policies together with an adaptive prefetching policy achieves a high hit ratio [7]. Different from our work, they consider caching and prefetching of pages of results.

As systems are often hierarchical, there has also been some effort on multi-level architectures. Saraiva *et al.* propose a new architecture for Web search engines using a two-level dynamic caching system [13]. Their goal for such systems has been to improve response time for hierarchical engines. In their architecture, both levels use an LRU eviction policy. They find that the second-level cache can effectively reduce disk traffic, thus increasing the overall throughput. Baeza-Yates and Saint-Jean propose a three-level index organization [2]. Long and Suel propose a caching system structured according to three different levels [9]. The intermediate level contains frequently occurring pairs of terms and stores the intersections of the corresponding inverted lists. These last two papers are related to ours in that they exploit different caching strategies at different levels of the memory hierarchy.

Finally, our static caching algorithm for posting lists in Section 5 uses the ratio frequency/size in order to evaluate the goodness of an item to cache. Similar ideas have been used in the context of file caching [17], Web caching [5], and even caching of posting lists [9], but in all cases in a dynamic setting. To the best of our knowledge we are the first to use this approach for static caching of posting lists.

## 3. DATA CHARACTERIZATION

Our data consists of a crawl of documents from the UK domain, and query logs of one year of queries submitted to http://www.yahoo.co.uk from November 2005 to November 2006. In our logs, 50% of the total volume of queries are unique. The average query length is 2.5 terms, with the longest query having 731 terms.

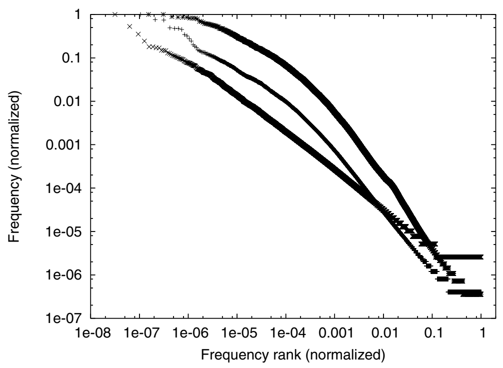

**Figure 2: The distribution of queries (bottom curve) and query terms (middle curve) in the query log. The distribution of document frequencies of terms in the UK-2006 dataset (upper curve).**

Figure 2 shows the distributions of queries (lower curve), and query terms (middle curve). The $x$-axis represents the normalized frequency rank of the query or term. (The most frequent query appears closest to the y-axis.) The $y$-axis is

**Table 1: Statistics of the UK-2006 sample.**

| UK-2006 sample statistics | |
|---|---|
| # of documents | 2,786,391 |
| # of terms | 6,491,374 |
| # of tokens | 2,109,512,558 |

the normalized frequency for a given query (or term). As expected, the distribution of query frequencies and query term frequencies follow power law distributions, with slope of 1.84 and 2.26, respectively. In this figure, the query frequencies were computed as they appear in the logs with no normalization for case or white space. The query terms (middle curve) have been normalized for case, as have the terms in the document collection.

The document collection that we use for our experiments is a summary of the UK domain crawled in May 2006.[1] This summary corresponds to a maximum of 400 crawled documents per host, using a breadth first crawling strategy, comprising 15GB. The distribution of document frequencies of terms in the collection follows a power law distribution with slope 2.38 (upper curve in Figure 2). The statistics of the collection are shown in Table 1. We measured the correlation between the document frequency of terms in the collection and the number of queries that contain a particular term in the query log to be 0.424. A scatter plot for a random sample of terms is shown in Figure 3. In this experiment, terms have been converted to lower case in both the queries and the documents so that the frequencies will be comparable.

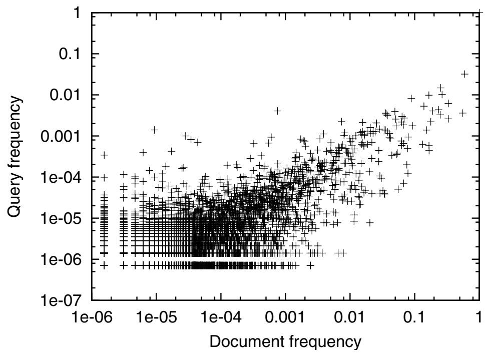

**Figure 3: Normalized scatter plot of document-term frequencies vs. query-term frequencies.**

## 4. CACHING OF QUERIES AND TERMS

Caching relies upon the assumption that there is locality in the stream of requests. That is, there must be sufficient repetition in the stream of requests and within intervals of time that enable a cache memory of reasonable size to be effective. In the query log we used, 88% of the unique queries are singleton queries, and 44% are singleton queries out of the whole volume. Thus, out of all queries in the stream composing the query log, the upper threshold on hit ratio is 56%. This is because only 56% of all the queries comprise queries that have multiple occurrences. It is important to observe, however, that not all queries in this 56% can be cache hits because of compulsory misses. A compulsory miss

[1]The collection is available from the University of Milan: http://law.dsi.unimi.it/. URL retrieved 05/2007.

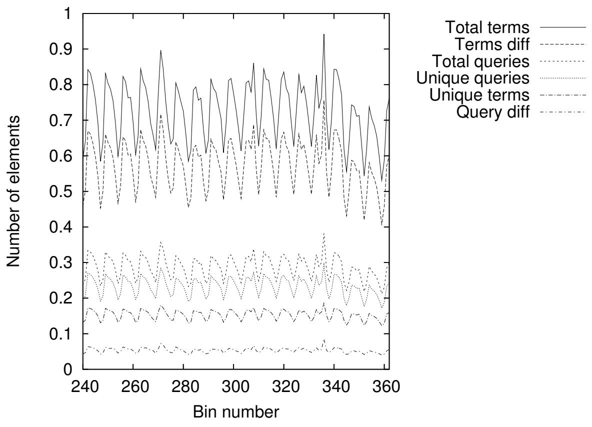

**Figure 4: Arrival rate for both terms and queries.**

happens when the cache receives a query for the first time. This is different from capacity misses, which happen due to space constraints on the amount of memory the cache uses. If we consider a cache with infinite memory, then the hit ratio is 50%. Note that for an infinite cache there are no capacity misses.

As we mentioned before, another possibility is to cache the posting lists of terms. Intuitively, this gives more freedom in the utilization of the cache content to respond to queries because cached terms might form a new query. On the other hand, they need more space.

As opposed to queries, the fraction of singleton terms in the total volume of terms is smaller. In our query log, only 4% of the terms appear once, but this accounts for 73% of the vocabulary of query terms. We show in Section 5 that caching a small fraction of terms, while accounting for terms appearing in many documents, is potentially very effective.

Figure 4 shows several graphs corresponding to the normalized arrival rate for different cases using days as bins. That is, we plot the normalized number of elements that appear in a day. This graph shows only a period of 122 days, and we normalize the values by the maximum value observed throughout the whole period of the query log. “Total queries” and “Total terms” correspond to the total volume of queries and terms, respectively. “Unique queries” and “Unique terms” correspond to the arrival rate of unique queries and terms. Finally, “Query diff” and “Terms diff” correspond to the difference between the curves for total and unique.

In Figure 4, as expected, the volume of terms is much higher than the volume of queries. The difference between the total number of terms and the number of unique terms is much larger than the difference between the total number of queries and the number of unique queries. This observation implies that terms repeat significantly more than queries. If we use smaller bins, say of one hour, then the ratio of unique to volume is higher for both terms and queries because it leaves less room for repetition.

We also estimated the workload using the document frequency of terms as a measure of how much work a query imposes on a search engine. We found that it follows closely the arrival rate for terms shown in Figure 4.

To demonstrate the effect of a dynamic cache on the query frequency distribution of Figure 2, we plot the same frequency graph, but now considering the frequency of queries

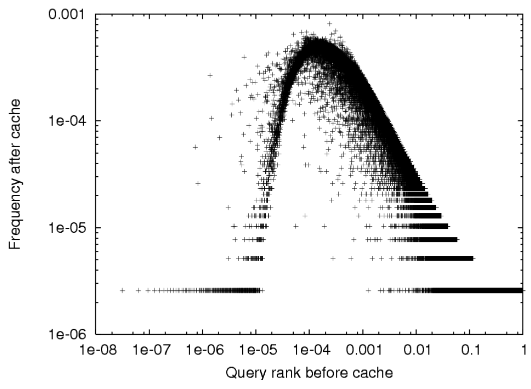

**Figure 5: Frequency graph after LRU cache.**

after going through an LRU cache. On a cache miss, an LRU cache decides upon an entry to evict using the information on the recency of queries. In this graph, the most frequent queries are not the same queries that were most frequent before the cache. It is possible that queries that are most frequent after the cache have different characteristics, and tuning the search engine to queries frequent before the cache may degrade performance for non-cached queries. The maximum frequency after caching is less than 1% of the maximum frequency before the cache, thus showing that the cache is very effective in reducing the load of frequent queries. If we re-rank the queries according to after-cache frequency, the distribution is still a power law, but with a much smaller value for the highest frequency.

When discussing the effectiveness of dynamically caching, an important metric is cache miss rate. To analyze the cache miss rate for different memory constraints, we use the *working set model* [6, 14]. A working set, informally, is the set of references that an application or an operating system is currently working with. The model uses such sets in a strategy that tries to capture the temporal locality of references. The working set strategy then consists in keeping in memory only the elements that are referenced in the previous $\theta$ steps of the input sequence, where $\theta$ is a configurable parameter corresponding to the window size.

Originally, working sets have been used for page replacement algorithms of operating systems, and considering such a strategy in the context of search engines is interesting for three reasons. First, it captures the amount of locality of queries and terms in a sequence of queries. Locality in this case refers to the frequency of queries and terms in a window of time. If many queries appear multiple times in a window, then locality is high. Second, it enables an offline analysis of the expected miss rate given different memory constraints. Third, working sets capture aspects of efficient caching algorithms such as LRU. LRU assumes that references farther in the past are less likely to be referenced in the present, which is implicit in the concept of working sets [14].

Figure 6 plots the miss rate for different working set sizes, and we consider working sets of both queries and terms. The working set sizes are normalized against the total number of queries in the query log. In the graph for queries, there is a sharp decay until approximately 0.01, and the rate at which the miss rate drops decreases as we increase the size of the working set over 0.01. Finally, the minimum value it reaches is 50% miss rate, not shown in the figure as we have cut the tail of the curve for presentation purposes.

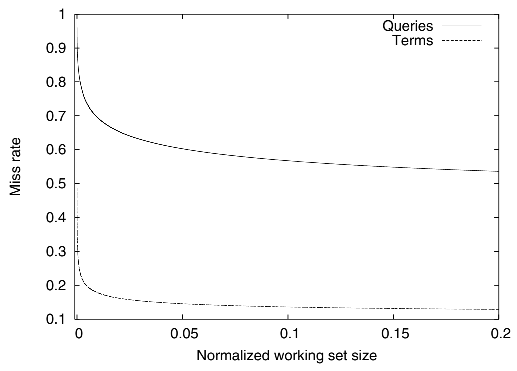

**Figure 6: Miss rate as a function of the working set size.**

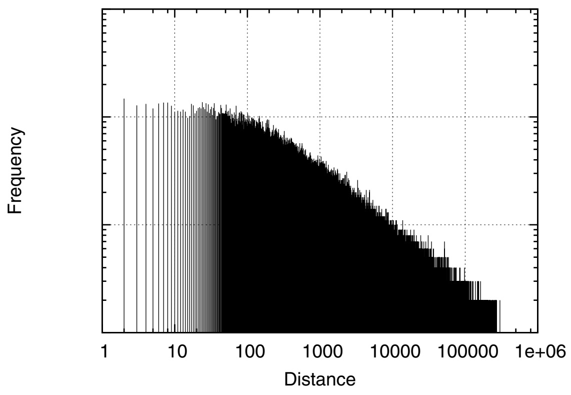

**Figure 7: Distribution of distances expressed in terms of distinct queries.**

Compared to the query curve, we observe that the minimum miss rate for terms is substantially smaller. The miss rate also drops sharply on values up to 0.01, and it decreases minimally for higher values. The minimum value, however, is slightly over 10%, which is much smaller than the minimum value for the sequence of queries. This implies that with such a policy it is possible to achieve over 80% hit rate, if we consider caching dynamically posting lists for terms as opposed to caching answers for queries. This result does not consider the space required for each unit stored in the cache memory, or the amount of time it takes to put together a response to a user query. We analyze these issues more carefully later in this paper.

It is interesting also to observe the histogram of Figure 7, which is an intermediate step in the computation of the miss rate graph. It reports the distribution of distances between repetitions of the same frequent query. The distance in the plot is measured in the number of distinct queries separating a query and its repetition, and it considers only queries appearing at least 10 times. From Figures 6 and 7, we conclude that even if we set the size of the query answers cache to a relatively large number of entries, the miss rate is high. Thus, caching the posting lists of terms has the potential to improve the hit ratio. This is what we explore next.

## 5. CACHING POSTING LISTS

The previous section shows that caching posting lists can obtain a higher hit rate compared to caching query answers. In this section we study the problem of how to select posting lists to place on a certain amount of available memory, assuming that the whole index is larger than the amount of memory available. The posting lists have variable size (in fact, their size distribution follows a power law), so it is beneficial for a caching policy to consider the sizes of the posting lists. We consider both *dynamic* and *static* caching. For dynamic caching, we use two well-known policies, LRU and LFU, as well as a modified algorithm that takes posting-list size into account.

Before discussing the static caching strategies, we introduce some notation. We use $f_q(t)$ to denote the query-term frequency of a term $t$, that is, the number of queries containing $t$ in the query log, and $f_d(t)$ to denote the document frequency of $t$, that is, the number of documents in the collection in which the term $t$ appears.

The first strategy we consider is the algorithm proposed by Baeza-Yates and Saint-Jean [2], which consists in selecting the posting lists of the terms with the highest query-term frequencies $f_q(t)$. We call this algorithm QTF.

We observe that there is a trade-off between $f_q(t)$ and $f_d(t)$. Terms with high $f_q(t)$ are useful to keep in the cache because they are queried often. On the other hand, terms with high $f_d(t)$ are not good candidates because they correspond to long posting lists and consume a substantial amount of space. In fact, the problem of selecting the best posting lists for the static cache corresponds to the standard KNAPSACK problem: given a knapsack of fixed capacity, and a set of $n$ items, such as the $i$-th item has *value* $c_i$ and *size* $s_i$, select the set of items that fit in the knapsack and maximize the overall value. In our case, "value" corresponds to $f_q(t)$ and "size" corresponds to $f_d(t)$. Thus, we employ a simple algorithm for the knapsack problem, which is selecting the posting lists of the terms with the highest values of the ratio $\frac{f_q(t)}{f_d(t)}$. We call this algorithm QTFDF. We tried other variations considering query frequencies instead of term frequencies, but the gain was minimal compared to the complexity added.

In addition to the above two static algorithms we consider the following algorithms for dynamic caching:

- LRU: A standard LRU algorithm, but many posting lists might need to be evicted (in order of least-recent usage) until there is enough space in the memory to place the currently accessed posting list;
- LFU: A standard LFU algorithm (eviction of the least-frequently used), with the same modification as the LRU;
- DYN-QTFDF: A dynamic version of the QTFDF algorithm; evict from the cache the term(s) with the lowest $\frac{f_q(t)}{f_d(t)}$ ratio.

The performance of all the above algorithms for 15 weeks of the query log and the UK dataset are shown in Figure 8. Performance is measured with *hit rate*. The cache size is measured as a fraction of the total space required to store the posting lists of all terms.

For the dynamic algorithms, we load the cache with terms in order of $f_q(t)$ and we let the cache "warm up" for 1 million queries. For the static algorithms, we assume complete knowledge of the frequencies $f_q(t)$, that is, we estimate $f_q(t)$ from the whole query stream. As we show in Section 7 the results do not change much if we compute the query-term frequencies using the first 3 or 4 weeks of the query log and measure the hit rate on the rest.

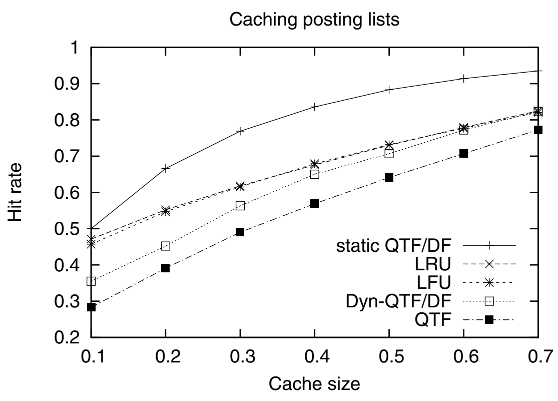

**Figure 8: Hit rate of different strategies for caching posting lists.**

The most important observation from our experiments is that the static QTFDF algorithm has a better hit rate than all the dynamic algorithms. An important benefit a static cache is that it requires no eviction and it is hence more efficient when evaluating queries. However, if the characteristics of the query traffic change frequently over time, then it requires re-populating the cache often or there will be a significant impact on hit rate.

## 6. ANALYSIS OF STATIC CACHING

In this section we provide a detailed analysis for the problem of deciding whether it is preferable to cache query answers or cache posting lists. Our analysis takes into account the impact of caching between two levels of the data-access hierarchy. It can either be applied at the memory/disk layer or at a server/remote server layer as in the architecture we discussed in the introduction.

Using a particular system model, we obtain estimates for the parameters required by our analysis, which we subsequently use to decide the optimal trade-off between caching query answers and caching posting lists.

### 6.1 Analytical Model

Let $M$ be the size of the cache measured in answer units (the cache can store $M$ query answers). Assume that all posting lists are of the same length $L$, measured in answer units. We consider the following two cases: $(A)$ a cache that stores only precomputed answers, and $(B)$ a cache that stores only posting lists. In the first case, $N_c = M$ answers fit in the cache, while in the second case $N_p = M/L$ posting lists fit in the cache. Thus, $N_p = N_c/L$. Note that although posting lists require more space, we can combine terms to evaluate more queries (or partial queries).

For case $(A)$, suppose that a query answer in the cache can be evaluated in 1 time unit. For case $(B)$, assume that if the posting lists of the terms of a query are in the cache then the results can be computed in $TR_1$ time units, while if the posting lists are not in the cache then the results can be computed in $TR_2$ time units. Of course $TR_2 > TR_1$.

Now we want to compare the time to answer a stream of $Q$ queries in both cases. Let $V_c(N_c)$ be the volume of the most frequent $N_c$ queries. Then, for case $(A)$, we have an overall time

$$T_{CA} = V_c(N_c) + TR_2(Q - V_c(N_c)).$$

Similarly, for case $(B)$, let $V_p(N_p)$ be the number of computable queries. Then we have overall time

$$T_{PL} = TR_1 V_p(N_p) + TR_2(Q - V_p(N_p)).$$

We want to check under which conditions we have $T_{PL} < T_{CA}$. We have

$$T_{PL} - T_{CA} = (TR_2 - 1)V_c(N_c) - (TR_2 - TR_1)V_p(N_p) > 0.$$

Figure 9 shows the values of $V_p$ and $V_c$ for our data. We can see that caching answers saturates faster and for this particular data there is no additional benefit from using more than 10% of the index space for caching answers.

As the query distribution is a power law with parameter $\alpha > 1$, the $i$-th most frequent query appears with probability proportional to $\frac{1}{i^\alpha}$. Therefore, the volume $V_c(n)$, which is the total number of the $n$ most frequent queries, is

$$V_c(n) = V_0 \sum_{i=1}^{n} \frac{Q}{i^\alpha} = \gamma_n Q \quad (0 < \gamma_n < 1).$$

We know that $V_p(n)$ grows faster than $V_c(n)$ and assume, based on experimental results, that the relation is of the form $V_p(n) = k\ V_c(n)^\beta$.

In the worst case, for a large cache, $\beta \to 1$. That is, both techniques will cache a constant fraction of the overall query volume. Then caching posting lists makes sense only if

$$\frac{L(TR_2 - 1)}{k(TR_2 - TR_1)} > 1.$$

If we use compression, we have $L' < L$ and $TR'_1 > TR_1$. According to the experiments that we show later, compression is always better.

For a small cache, we are interested in the transient behavior and then $\beta > 1$, as computed from our data. In this case there will always be a point where $T_{PL} > T_{CA}$ for a large number of queries.

In reality, instead of filling the cache only with answers or only with posting lists, a better strategy will be to divide the total cache space into cache for answers and cache for posting lists. In such a case, there will be some queries that could be answered by both parts of the cache. As the answer cache is faster, it will be the first choice for answering those queries. Let $Q_{N_c}$ and $Q_{N_p}$ be the set of queries that can be answered by the cached answers and the cached posting lists, respectively. Then, the overall time is

$$T = V_c(N_c) + TR_1 V(Q_{N_p} - Q_{N_c}) + TR_2(Q - V(Q_{N_p} \cup Q_{N_c})),$$

where $N_p = (M - N_c)/L$. Finding the optimal division of the cache in order to minimize the overall retrieval time is a difficult problem to solve analytically. In Section 6.3 we use simulations to derive optimal cache trade-offs for particular implementation examples.

## 6.2 Parameter Estimation

We now use a particular implementation of a centralized system and the model of a distributed system as examples from which we estimate the parameters of the analysis from the previous section. We perform the experiments using an optimized version of Terrier [11] for both indexing documents and processing queries, on a single machine with a Pentium 4 at 2GHz and 1GB of RAM.

We indexed the documents from the UK-2006 dataset, without removing stop words or applying stemming. The posting lists in the inverted file consist of pairs of document identifier and term frequency. We compress the document identifier gaps using Elias gamma encoding, and the term frequencies in documents using unary encoding [16]. The size of the inverted file is 1,189Mb. A stored answer requires 1264 bytes, and an uncompressed posting takes 8 bytes. From Table 1, we obtain $L = \frac{(8 \cdot \#\text{ of postings})}{1264 \cdot \#\text{ of terms}} = 0.75$ and $L' = \frac{\text{Inverted file size}}{1264 \cdot \#\text{ of terms}} = 0.26$.

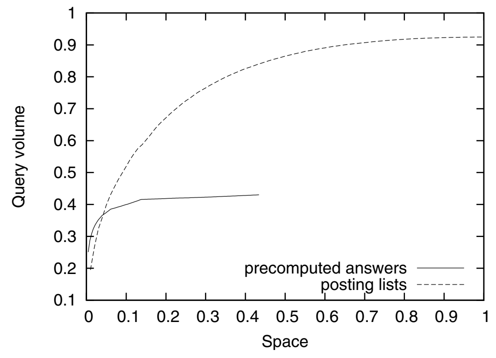

**Figure 9: Cache saturation as a function of size.**

**Table 2: Ratios between the average time to evaluate a query and the average time to return cached answers (centralized and distributed case).**

| Centralized system | $TR_1$ | $TR_2$ | $TR'_1$ | $TR'_2$ |
|---|---|---|---|---|
| Full evaluation | 233 | 1760 | 707 | 1140 |
| Partial evaluation | 99 | 1626 | 493 | 798 |
| **LAN system** | $TR^L_1$ | $TR^L_2$ | $TR'^L_1$ | $TR'^L_2$ |
| Full evaluation | 242 | 1769 | 716 | 1149 |
| Partial evaluation | 108 | 1635 | 502 | 807 |
| **WAN system** | $TR^W_1$ | $TR^W_2$ | $TR'^W_1$ | $TR'^W_2$ |
| Full evaluation | 5001 | 6528 | 5475 | 5908 |
| Partial evaluation | 4867 | 6394 | 5270 | 5575 |

We estimate the ratio $TR = T/T_c$ between the average time $T$ it takes to evaluate a query and the average time $T_c$ it takes to return a stored answer for the same query, in the following way. $T_c$ is measured by loading the answers for 100,000 queries in memory, and answering the queries from memory. The average time is $T_c = 0.069ms$. $T$ is measured by processing the same 100,000 queries (the first 10,000 queries are used to warm-up the system). For each query, we remove stop words, if there are at least three remaining terms. The stop words correspond to the terms with a frequency higher than the number of documents in the index. We use a document-at-a-time approach to retrieve documents containing all query terms. The only disk access required during query processing is for reading compressed posting lists from the inverted file. We perform both full and partial evaluation of answers, because some queries are likely to retrieve a large number of documents, and only a fraction of the retrieved documents will be seen by users. In the partial evaluation of queries, we terminate the processing after matching 10,000 documents. The estimated ratios $TR$ are presented in Table 2.

Figure 10 shows for a sample of queries the workload of the system with partial query evaluation and compressed posting lists. The x-axis corresponds to the total time the system spends processing a particular query, and the vertical axis corresponds to the sum $\sum_{t \in q} f_q \cdot f_d(t)$. Notice that the total number of postings of the query-terms does not necessarily provide an accurate estimate of the workload imposed on the system by a query (which is the case for full evaluation and uncompressed lists).

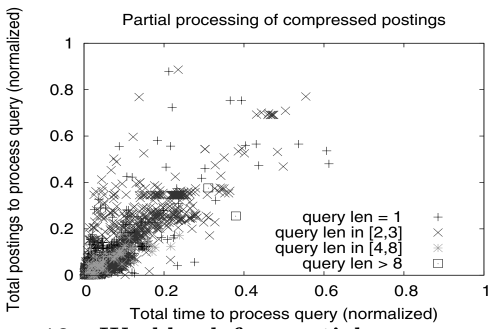

**Figure 10: Workload for partial query evaluation with compressed posting lists.**

The analysis of the previous section also applies to a distributed retrieval system in one or multiple sites. Suppose that a document partitioned distributed system is running on a cluster of machines interconnected with a local area network (LAN) in one site. The broker receives queries and broadcasts them to the query processors, which answer the queries and return the results to the broker. Finally, the broker merges the received answers and generates the final set of answers (we assume that the time spent on merging results is negligible). The difference between the centralized architecture and the document partition architecture is the extra communication between the broker and the query processors. Using ICMP pings on a 100Mbps LAN, we have measured that sending the query from the broker to the query processors which send an answer of 4,000 bytes back to the broker takes on average 0.615ms. Hence, $TR^L = TR + 0.615ms/0.069ms = TR + 9$.

In the case when the broker and the query processors are in different sites connected with a wide area network (WAN), we estimated that broadcasting the query from the broker to the query processors and getting back an answer of 4,000 bytes takes on average 329ms. Hence, $TR^W = TR + 329ms/0.069ms = TR + 4768$.

## 6.3 Simulation Results

We now address the problem of finding the optimal trade-off between caching query answers and caching posting lists. To make the problem concrete we assume a fixed budget $M$ on the available memory, out of which $x$ units are used for caching query answers and $M - x$ for caching posting lists.

We perform simulations and compute the average response time as a function of $x$. Using a part of the query log as training data, we first allocate in the cache the answers to the most frequent queries that fit in space $x$, and then we use the rest of the memory to cache posting lists. For selecting posting lists we use the QtfDf algorithm, applied to the training query log but excluding the queries that have already been cached.

In Figure 11, we plot the simulated response time for a centralized system as a function of $x$. For the uncompressed index we use $M = 1$GB, and for the compressed index we use $M = 0.5$GB. In the case of the configuration that uses partial query evaluation with compressed posting lists, the lowest response time is achieved when 0.15GB out of the 0.5GB is allocated for storing answers for queries. We obtained similar trends in the results for the LAN setting.

Figure 12 shows the simulated workload for a distributed system across a WAN. In this case, the total amount of memory is split between the broker, which holds the cached answers of queries, and the query processors, which hold the cache of posting lists. According to the figure, the difference between the configurations of the query processors is less important because the network communication overhead increases the response time substantially. When using uncompressed posting lists, the optimal allocation of memory corresponds to using approximately 70% of the memory for caching query answers. This is explained by the fact that there is no need for network communication when the query can be answered by the cache at the broker.

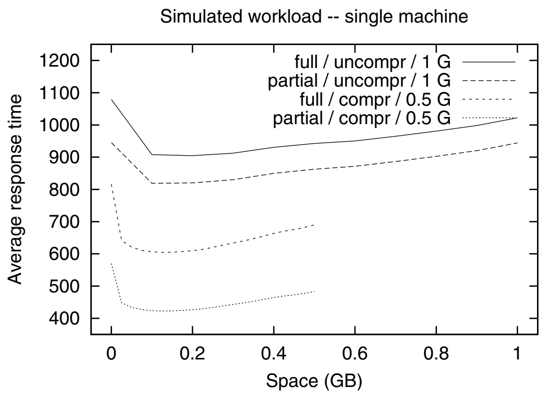

**Figure 11: Optimal division of the cache in a server.**

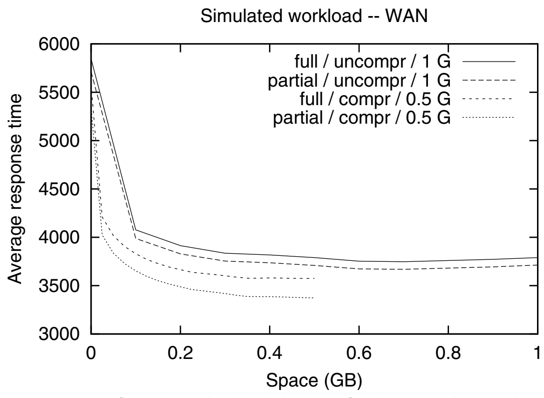

**Figure 12: Optimal division of the cache when the next level requires WAN access.**

## 7. EFFECT OF THE QUERY DYNAMICS

For our query log, the query distribution and query-term distribution change slowly over time. To support this claim, we first assess how topics change comparing the distribution of queries from the first week in June, 2006, to the distribution of queries for the remainder of 2006 that did not appear in the first week in June. We found that a very small percentage of queries are new queries. The majority of queries that appear in a given week repeat in the following weeks for the next six months.

We then compute the hit rate of a static cache of 128,000 answers trained over a period of two weeks (Figure 13). We report hit rate hourly for 7 days, starting from 5pm. We observe that the hit rate reaches its highest value during the night (around midnight), whereas around 2-3pm it reaches its minimum. After a small decay in hit rate values, the hit rate stabilizes between 0.28, and 0.34 for the entire week, suggesting that the static cache is effective for a whole week after the training period.

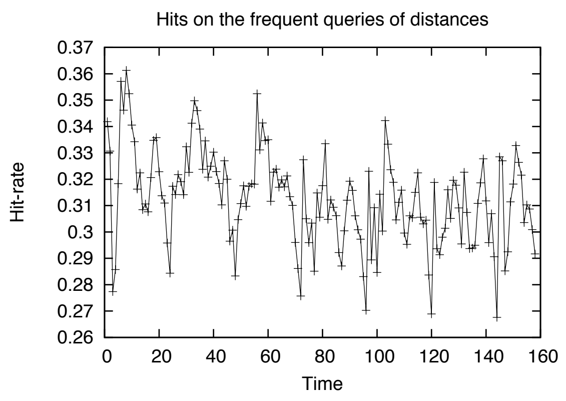

**Figure 13: Hourly hit rate for a static cache holding 128,000 answers during the period of a week.**

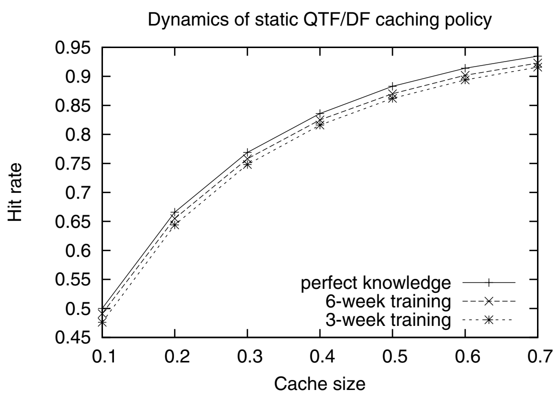

**Figure 14: Impact of distribution changes on the static caching of posting lists.**

The static cache of posting lists can be periodically recomputed. To estimate the time interval in which we need to recompute the posting lists on the static cache we need to consider an efficiency/quality trade-off: using too short a time interval might be prohibitively expensive, while recomputing the cache too infrequently might lead to having an obsolete cache not corresponding to the statistical characteristics of the current query stream.

We measured the effect on the QtfDf algorithm of the changes in a 15-week query stream (Figure 14). We compute the query term frequencies over the whole stream, select which terms to cache, and then compute the hit rate on the whole query stream. This hit rate is as an upper bound, and it assumes perfect knowledge of the query term frequencies. To simulate a realistic scenario, we use the first 6 (3) weeks of the query stream for computing query term frequencies and the following 9 (12) weeks to estimate the hit rate. As Figure 14 shows, the hit rate decreases by less than 2%. The high correlation among the query term frequencies during different time periods explains the graceful adaptation of the static caching algorithms to the future query stream. Indeed, the pairwise correlation among all possible 3-week periods of the 15-week query stream is over 99.5%.

## 8. CONCLUSIONS

Caching is an effective technique in search engines for improving response time, reducing the load on query processors, and improving network bandwidth utilization. We present results on both dynamic and static caching. Dynamic caching of queries has limited effectiveness due to the high number of compulsory misses caused by the number of unique or infrequent queries. Our results show that in our UK log, the minimum miss rate is 50% using a working set strategy. Caching terms is more effective with respect to miss rate, achieving values as low as 12%. We also propose a new algorithm for static caching of posting lists that outperforms previous static caching algorithms as well as dynamic algorithms such as LRU and LFU, obtaining hit rate values that are over 10% higher compared these strategies.

We present a framework for the analysis of the trade-off between caching query results and caching posting lists, and we simulate different types of architectures. Our results show that for centralized and LAN environments, there is an optimal allocation of caching query results and caching of posting lists, while for WAN scenarios in which network time prevails it is more important to cache query results.

## 9. REFERENCES

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *ACM CIKM*, 2006.
[2] R. A. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, 2003.
[3] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *ACM SIGIR*, 1985.
[4] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *ACM CIKM*, 2006.
[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USITS*, 1997.
[6] P. Denning. Working sets past and present. *IEEE Trans. on Software Engineering*, SE-6(1):64–84, 1980.
[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. Inf. Syst.*, 24(1):51–78, 2006.
[8] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, 2003.
[9] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, 2005.
[10] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.
[11] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *SIGIR Workshop on Open Source Information Retrieval*, 2006.
[12] V. V. Raghavan and H. Sever. On the reuse of past optimal queries. In *ACM SIGIR*, 1995.
[13] P. C. Saraiva, E. S. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *ACM SIGIR*, 2001.
[14] D. R. Slutz and I. L. Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, 1974.
[15] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *ACM SIGIR*, 2005.
[16] I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, Inc., NY, 1994.
[17] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.