Two Level Caching Techniques for Improving Result Ranking

Amarjeet Singh[1], **Mohd. Hussain**[2] and **Rakesh Ranjan**[3]

Submitted in November 2010, Accepted in July 2011

Abstract - ***Due to the rapid growth of the Web from a few thousand pages in 2000 to its current size of several billion pages, users increasingly depend on web search engines for locating relevant information. One of the main challenges for search engines is to provide a good ranking function that can identify the most useful results from among the many relevant pages, and a lot of research has focused on how to improve ranking, We present an effective caching scheme that reduces the computing and I/O requirements of a Web search engine without altering its ranking characteristics. The novelty is a two-level caching scheme that simultaneously combines cached query results and cached inverted lists on a real case search engine. A set of log queries are used to measure and compare the performance and the scalability of the search engine with no cache, with the cache for query results, with the cache for inverted lists, and with the two-level cache. Experimental results show that the two-level cache is superior, and that it allows increasing the maximum number of queries processed per second by a factor of three, while preserving the response time.***

Index Terms: Search Engines, Query Processing, Retrieval, Ranking, Cache Design

1. INTRODUCTION

Large web search engines have to answer thousands of queries per second with interactive response times. Due to the sizes of the data sets involved, often in the range of multiple terabytes, a single query may require the processing of hundreds of megabytes or more of index data. To keep up with this immense workload, large search engines employ clusters of hundreds or thousands of machines, and a number of techniques such as caching, index compression, and index and query pruning are used to improve scalability. In particular, two-level caching techniques cache results of repeated identical queries at the frontend, while index data for frequently used query terms are cached in each node at a lower level. Popular search engines receive millions of queries daily, a load never experienced before by any IR system. Additionally, search engines have to deal with a growing number of Web pages to discover, to index and to retrieve, and must handle very large databases. To compound the problem, search engine users want to experience small response times as well as precise and relevant results for their queries. In this scenario, the development of techniques to improve the performance and the scalability of search engines without degrading the quality of the results becomes a fundamental topic of research in IR. One effective alternative for improving performance and scalability of information systems is caching. The effectiveness of caching strategies depends on some key aspects, such as the presence of reference locality in the access stream and the frequency at which the database being cached is updated.

In this paper we describe and evaluate the implementation of caching schemes that improve the scalability of search engines without altering their ranking characteristics. The starting point of the work is TodoBR, a state-of-the-art full scale operational search engine that crawls the Brazilian Web. We enhanced the current implementation of TodoBR by integrating three caching schemes. The first one implements a cache of query results, allowing the search engine to answer recently repeated queries at a very low cost, since it is not necessary to process those queries. The second one implement a cache of the inverted lists of query terms, thus improving the query processing time for the new queries that include at least one term whose list is cached. The third caching scheme combines the two previous approaches and will be called two-level cache.

Each of the first two strategies presents advantages and disadvantages. A hit in the cache of query results avoids query processing, while a hit in the cache of inverted lists reduces the amount of I/O associated with answering a query, but does not avoid the query processing costs. On the other hand, the hit ratio associated with inverted lists is usually higher than the hit ratio for whole queries, which may pay o_ the query processing cost. The motivation behind the third strategy is to exploit the advantages of the first two strategies to improve even further the overall performance and scalability of the search engines.

Our experimental evaluation yields some key results. The two-level cache is superior and allows increasing the maximum throughput by a factor of three, relative to an implementation with no cache. Furthermore, the throughput of the two-level cache is up to 52% higher than the implementation using just cache of inverted lists and up to 36% higher than the cache of query results. Our work is distinct from previous ones because it presents experimental results on the effectiveness of different caching strategies implemented on a real case search engine.

E-Mail: [1]*amarjeetsingh_9@rediffmail.com,*
[2]*mohd.husain90@gmail.com and*
[3]*rakeshranjan.lko@gmail.com*

Our main contribution is the two-level caching scheme we proposed which yields superior performance. Our results can be replicated to other Web search engines since there is high similarity between workload characteristics present in the logs of TodoBR search engine and in the logs of other large search engines.

2. SEARCH ENGINE ARCHITECTURE

Web search engines are IR systems that take a query as input and produce as a result a set of links to relevant Web pages related to the query. Search engines seek, collect and index Web pages on a massive scale. To speed up query processing, all queries are answered using the index and without accessing the text directly.

Efficient query evaluation requires specialized index techniques when the text collection is large. Our search engine server implementation uses an inverted file as index structure, a popular choice to implement large scale IR systems. An inverted file is typically composed of a vocabulary, which contains the set of all distinct terms in the collection, and an inverted list for each term of the vocabulary. The inverted list of a term t is a list of the identifiers of the documents containing t with the respective frequency of occurrences of t on each document.

The ranking method used for the experiments is based on the vector space model. In the vector space model, the documents and the queries are represented as vectors in a space with dimensions given by the size of the vocabulary. The answers to the queries are the documents with the highest similarity values, where the similarity is computed by the cosine of the angle between the query vector and each document vector. The inverted file is used during query processing time to compute the similarities of each document of the collection against the query.

For large document databases, the cost of evaluating the cosine measure may be potentially high, because it assigns a similarity measure to every document containing any of the query terms, requiring a read and some processing on the whole inverted list of each term of the query. This task may be expensive since some of the terms can occur in a high proportion of the documents present in the database.

An effective technique to compute an approximation of the cosine measure without significant changes in the final ranking for each query is already proposed. We use it to process the queries submitted to the search engine server. This query evaluation technique uses early recognition of which documents are likely to be highly ranked to reduce costs of query processing. Queries are evaluated in 2% of the memory of the standard cosine implementation without degradation in retrieval effectiveness. Disk traffic and CPU time are also reduced because the algorithm processes only portions of the inverted lists which have information that can change the ranking.

3. CACHE DESIGN

In this section, we describe in detail the strategies for implementing the three caches in a search engine, that is, caching of query results, caching of inverted lists, and a two-level cache that combines both.

3.1 Cache of Query Results

Our strategy for caching query results is to keep in memory the list of documents associated with a given query. For each document we store its URL, its title, and a 250 character abstract. The very first implementation issue of this caching strategy is determining the number of document references that should be cached for each query. It is remarkable that the number of documents that match a given query is often huge.

However, the great majority of the users request at most the first 30 references that match a query. In TodoBR we also observe the same behavior, since most of the users (70%) do not request more than 10 references, and 90% of the query requests are for at most the first 50 references. Thus, we limited our cache of query results to 50 references, resulting in a storage requirement of 25 kilobytes per query result cached. This implementation decision allows our cache to satisfy most of the queries without wasting memory, and also exploits the spatial locality among queries. Figure 1 (a) shows the architecture of the search engine including the cache of query results. Whenever a user submits a query to the search engine, it checks whether the cache is storing the associated query results and the reference rank is below the caching threshold, in our case 50. If there is a cache hit, the query result is immediately returned to the user, at a very low cost, since the response only needs to be formatted and sent to the user, a cost inherent to any query. Otherwise, the search engine processes the query normally, occasionally caching it, whenever the reference rank is below the threshold.

The second major issue is the replacement policy for the query results, that is, how we determine which query results should be evicted from the cache whenever a new set of results is to be cached and the cache is full. In this first implementation we adopted LRU (least recently used) as replacement policy, since the TodoBR logs present a good temporal locality. Markatos has proposed alternative cache replacement policies for caching query results, such as SLRU (segmented LRU) and FBR (frequency based replacement), but they did not improve the cache hit ratio significantly. Furthermore, Markatos did not exploit spatial locality in his work, in the sense that a query result for the first ten documents is handled independently from the result for the next ten documents of the same query.

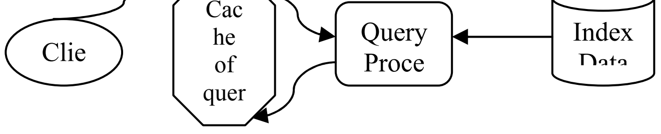


Figure 1(a): Query Results

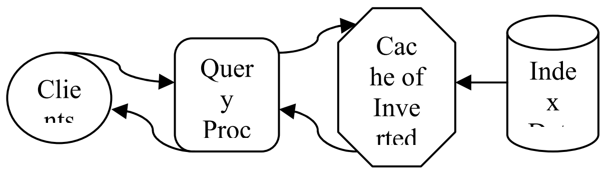


Figure 1(b): Inverted Lists

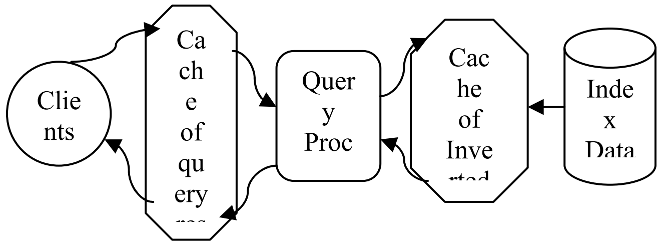


Figure 1(c): Two Levels

3.2 Cache of Inverted Lists

Our strategy for caching inverted lists is to keep in memory the list of Web documents associated with a given query term. In practice, our enhanced search engine caches the inverted lists for each term as they are accessed, and uses these lists to answer further queries that contain the same terms. In this case, the integration with the search engine is straightforward, since it acts as a specialized buffer for the index, which is usually stored in secondary memory. The main motivation for caching inverted lists is the good reference locality that is usually observed among individual search terms. Since the term locality is even greater than the query locality, and thus may attain a higher cache hit ratio, caching inverted lists is a good strategy for improving the scalability of search engines. The implementation of caches of inverted lists has to face two issues related to the high variance in the size of the inverted lists: the size of the cached lists and the internal organization of the cache.

These issues are discussed in the remaining of this section. The size of the inverted lists is a function of both the term popularity in the collection and the number of documents being indexed. For large collections, these lists may also become very large, making cache of inverted lists to fail in practice, since they require considerable cache space to store the whole list. To address this problem, we turn to an important characteristic of the filtered vector model processing technique. In this technique, the inverted lists are sorted by the frequency of occurrence of the term in each document, and the query processing exploits the frequency variance by using just the documents in which the term is most frequent. As a consequence, the lists are not fully traversed or are not traversed at all, depending on the relevance of the term on the collection and on the query it. In summary, the vector model allows naturally handling the problem associated with large inverted lists.

Since lists are almost always partially processed, we set out to cache parts of lists. The frequency-sorted inverted lists can be partitioned in different ways. The lists are naturally divided into blocks of documents in which the term appears with the same frequency, and these are the smallest units of algorithm processing. These blocks present interesting properties regarding their size and access pattern. The first blocks of each list are small, consisting of few documents, and are much more frequently accessed than the blocks at the end of the lists, which contain the documents in which the term appears a few times. In the model, given an inverted list of a term t, for some integer v (usually 2 to 4), a fraction (v - 1)/v of the document identifiers have frequency 1 ($f_{d.t}$ = 1); of the remainder a fraction (v - 1)/v have fd.t = 2, and so on. If v is 2, for example, half of the list will correspond to the block of documents in which the term appears only once. Blocks could be the objects to be cached, but their size distribution spans several orders of magnitude, making caching much more complex. Since the objects cached by a Web cache (html files, images, etc), also present extremely high variable sizes.

Using blocks as cacheable objects presents some advantages, but requires prefetching strategies and specific admission and replacement policies. For example, the first blocks of the lists tend to be very small and are generally accessed together. If no prefetching is done when the first block of a list is requested by the cache to the disk, there is a large number of disk seek operations to retrieve several small objects.

Another issue arises when the cache requests the last block of some large list. This is likely to be a large block, and its admission into the cache could cause the eviction of several other smaller but much more accessed blocks. These mechanisms and policies are certainly worthy of further study, but in this work we conjecture that much of the advantages of caching blocks can be attained by using a simpler alternative approach, namely to "page" the lists, i.e., to divide them into equally sized pages. We should observe that, based on the aforementioned distribution of sizes of blocks, the first pages of an inverted list may contain several blocks, while the last blocks of the list may span several pages. In this work we employed a page of 4 kilobytes which is also the disk block size. In our implementation, the cache only has knowledge of pages, and this makes for much simpler cache design. Furthermore, by varying the size of the pages, we can balance the tradeoff between the number of seek operations and the volume of bytes transferred from the disk. At one extreme, in which each byte of the inverted list is considered to be a page, there will be at least as many misses in the cache as the amount of bytes needed to answer a given workload of queries. The number of seek operations is maximal, while the volume of bytes transferred is minimal.

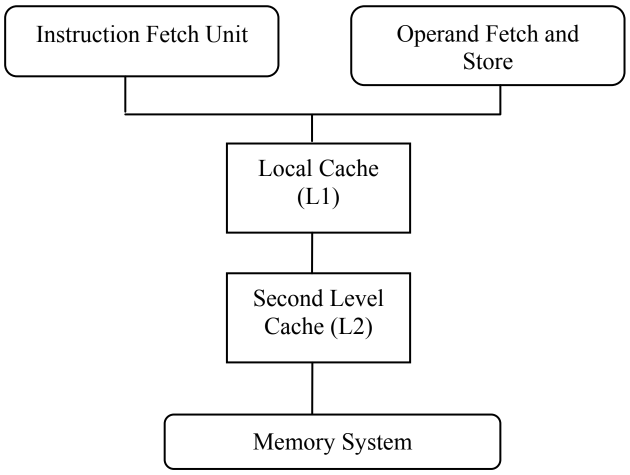
At the other extreme we consider a large page size, such that each list requires at most one miss in the cache. In this case, the number of seek operations is minimal, but the volume of bytes transferred is much larger than what is needed to answer the queries. Large pages have an amortizing effect on the disk seek time, and implicitly exploit spatial locality among list blocks, but may, on the other hand, cause the cache to store irrelevant parts of lists. Depending on the combination of factors, such as the costs associated with a disk seek operation and with the transferring of a byte, one can find an optimal page size. Other factors that should be taken into consideration are the disk block size and some operating system cache in effect. Figure 1 (b) illustrates the architecture of a search engine that embeds the cache of inverted lists. The query is processed as in the implementation with no cache up to a request to read a block, which is mapped to a page, from the inverted list, when the cache is checked. The disk is accessed only in the case of a miss in the cache of inverted lists. Again, we employed LRU as replacement policy. Although the cache of inverted lists avoids disk accesses, every query submitted to the system must still be processed, and gains in performance depend on the computational platform where the search engine runs.

3.3 Two-Level Cache

As discussed in the previous sections, each of the two cache architectures presents advantages and disadvantages. The cache of query results avoids processing queries which are already in the cache, while a hit in the cache of inverted lists only avoids disk accesses. On the other hand, the hit ratios obtained for the query results are smaller than the hit ratios obtained by the cache of inverted lists. These observations led us propose and test a third cache option, which combines the two caching strategies. We call this option two-level cache.

Figure 1 (c) shows the architecture of the search engine with a two-level cache system. Each request for the search engine is checked first in the cache of query results. If it is a hit, the query is answered immediately, otherwise the query is processed and the cache of inverted lists is used to reduce the number of disk accesses.

a. The L1 cache receives addresses from the prefetch and returns instructions either from the cache or from the next level of the memory hierarchy. The cache also receives addresses from the execution unit and reads or writes operands, again from the cache or from the next level of the hierarchy. The handling of writes varies with different write algorithms. If separate Ll instruction and data caches are present, they respond to the instruction fetch and instruction execution units, respectively.
b. The L2 cache receives addresses from the Ll cache (or caches) and reads or writes operands from its storage or from the primary memory system. The handling of writes varies with different write algorithms.



Basic Two-Level Simulation Model

1. The bus is a half-duplex data path connecting the caches to the memory system. Devices on the bus must arbitrate for bus ownership before commands or data can be sent.
2. The primary memory consists of a number of interleaved memories. Simulation parameters include the interleaving factor, access time, and cycle time of main memory.

4. WORKLOAD CHARACTERIZATION

In order to assess the behavior of the three cache implementations we consider in this paper, we perform an analysis of a partial log of queries submitted to TodoBR, comprising 100,256 queries. There is a total of 37,450 unique queries, and 23,751 unique terms in the log. We focus on aspects relevant to both levels of caching we consider, namely the characteristics of the stream of queries present in the log relevant to the cache of query results and of the stream of page references generated by the query processor - influencing the behavior of the cache of inverted lists.

In the case of the cache of inverted lists, we study its behavior under two different workloads, the first one with all the queries, and the second one with only the unique queries. To understand the reasons for this consideration, let us examine what happens to the cache of inverted lists under different configurations of the cache of query results. When used stand alone, the cache of inverted lists receives from the query processor a page workload originated from all of the queries received by the search engine. This is precisely the workload represented by the `All Queries' workload.

On the other hand, suppose a two-level implementation in which the cache of query results is large enough not to have any miss caused by eviction from the cache, i.e., it can store the results of every query that it receives. In this situation, the query processor, and thus the cache of inverted lists, will only process the unique queries, for all the repetitions will be handled by the cache of query results. The workload the cache of inverted lists will be subject to is well represented by the

`Unique Queries' workload. There will be a smooth transition from one workload to the other for varying sizes of the cache of query results, meaning that we can have valuable insight of the performance of the cache of inverted lists for a wide range of situations. A very small cache of query results will generate a workload at the cache of inverted lists similar to the `All Queries' workload, while a large cache of query results will generate a workload close to the `Unique Queries' workload.

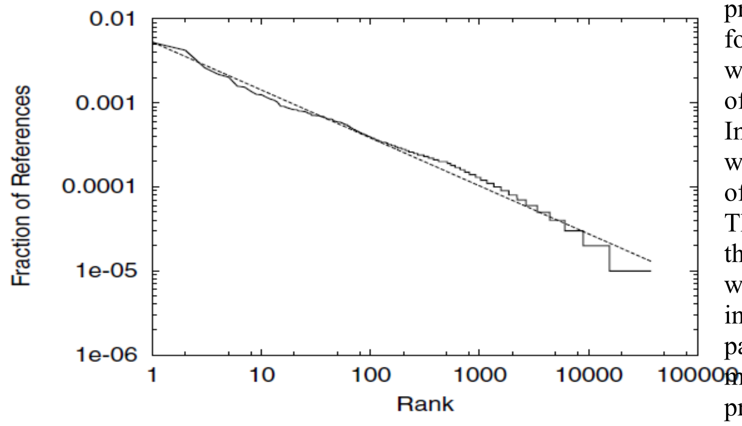


Figure2

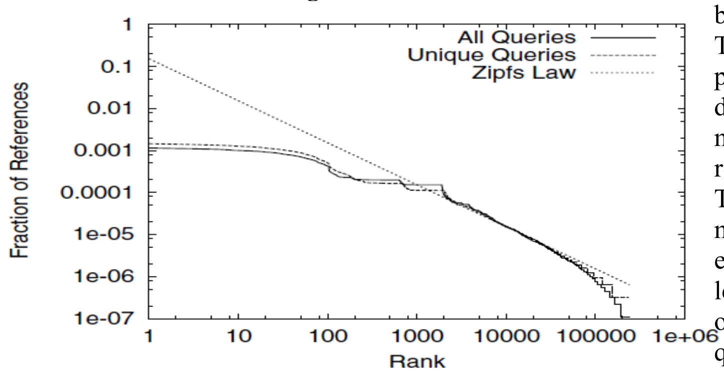


Figure3

4.1 Popularity

We start our workload characterization by analyzing the popularity of both queries and pages of the cache of inverted lists. We define popularity of an object as the number of references to the object, and the popularity rank as a list of all objects sorted by decreasing popularity, that is, the most popular object is the first in the rank.

For a reference stream to order good opportunity for caching, it ought to exhibit temporal locality among its references. In fact, the authors conclude that popularity is the main source of locality, specially in dealing with reasonably sized caches, and that a reference stream whose objects popularity follow a Zipf-like distribution exhibit a high degree of temporal locality. Zipf's law relates the popularity rank p of an object, to the probability P that it is requested, by P ~ 1/p, and has been applied to several distinct contexts, such as words in natural language and accesses to web pages. We call a Zipf-like popularity distribution the one in which the relation between P and p is given by $P \sim 1/p^{\alpha}$. This is a generalization of Zipf's law and in a log-log plot of popularity versus rank appears as a straight line with slope - α. The smaller α is, the less skewed the distribution is, showing weaker temporal locality and worse cache ability.

We verified that the references to queries follow a Zipf-like distribution. In Figure 2 we plot the relative popularity, i.e., the probability of accessing each query, versus the popularity rank for the queries stream, together with a Zipf-like distribution with an α parameter of 0:59, obtained by a least-squares fitting of the data.

In Figure 3 we examine the popularity distribution for both workloads of the cache of inverted lists. We can notice a pair of similar curves, labeled `All Queries' and `Unique Queries'. There are two regions in these two curves, one up to roughly the rank 2,500, with large at segments, and one after this point, which is approximately an straight line in the log-log plot with inclination of -1. The flat region occurs due to the page access pattern. The first pages of each list are accessed in group, meaning that they should have approximately the same probability of being accessed. This suggests, for caching effects, that the pages making up at region should necessarily be stored in the cache if it is to have a good level of efficiency. The second region, which comprises more than 90% of the pages, exhibit a Zipf-like behavior, and is well fit by one such distribution with α = 1. This indicates that the distributions much more skewed than that of the queries' popularities, resulting in greater temporal locality.

The distribution does not vary much for both workloads, meaning that there is opportunity for caching inverted lists even if this caching is to be done after a fully efficient first level cache of query results. In order to further investigate this opportunity, we collected statistics of the number of distinct queries in which each term appears. In the situation of a fully effective cache of query results, resulting in the `Unique Queries' workload to the cache of inverted lists, the terms that appear in only one query shall not generate a hit, because their pages will only be seen once by the cache of inverted lists. We found out that approximately 40% of the terms appear in more than one query, evidencing the extra locality that can be exploited by the cache of inverted lists.

4.2 Cache Miss Ratios

To assess the behavior of a cache under a LRU replacement policy, we generated the successive stack distances from the log. The marginal distribution of stack distances can be used to determine the miss ratio for a cache at different sizes. Let D be the random variable corresponding to stack distance, and let FD be the cumulative distribution function for D. The miss ratio m(x) for a cache holding x objects is given by

$$P[D > x] = 1 - F_{\alpha}(x) = m(x)$$

The first observation from the graph is the minimum miss ratio we can obtain under this query workload, which is around

40%. This is the miss ratio that an infinite cache would exhibit, and is due to the first occurrence of each query. The most important fact the graph shows is how fast the miss ratio decreases as we increase the capacity of the cache, relative to the TodoBR log we considered. We can observe a 'knee' in the curve close to 10 megabytes, indicating that a relatively small fraction of the queries accounts for a significant portion of the accesses.

This is a good indicator of the cache size that offers a good compromise between space and hit ratio. After this point, small decreases in the miss ratio come at the expense of large increases in cache size. It is with these considerations that we choose, for the following experiments, a cache size of 20 megabytes for query results. We point out that the fact that a cache of this size

holds most of the working set of the workload is much more important than the size itself, which should be determined in a case by case basis, by analyzing the miss ratio curve for the workload.

We can see similar miss ratio versus cache size curves for the cache of inverted lists under the two workloads considered. One can notice that the cache size at which there is a significant decrease in the miss ratio is much larger than in the case of the cache of query results, suggesting that the working set of the pages requires more cache space.

However the asymptotic miss ratio observed is much lower in the case of the cache of inverted lists, even for the `Unique Queries' workload. This shows the greater temporal locality present in the reference to pages, as was inferred from the popularity distributions. The miss ratio of the `All Queries' workload is considerably lower than the one of the `Unique Queries' workload, because in the latter only the repetition of terms across different queries do cause hits at the cache. Still, a 250 megabytes cache of inverted lists subject to the `Unique Queries' workload, i.e., the worst case workload for the second level cache, can achieve hit ratios of 80% on top of the misses at the first level.

We have a final word on the scalability of the characteristics presented herein. As we increase the length of the request stream submitted to the cache, the popularity distribution of queries and thus the marginal distribution of stack distances tend not to change much, meaning that a relatively small cache size should still be effective. Furthermore, the miss ratio tends to decrease as we increase the length of the request stream.

5. EXPERIMENTAL RESULTS

We present in this section experimental results that show the practical impact of the three caching schemes discussed on the scalability and on the average response time of the search engine as a whole.The experimental environment comprises two machines running Linux operating system version 2.2.16. The search engine runs on a Pentium III 550 MHz machine with 512 megabytes of main memory, and a 36 gigabytes SCSI disk. The client runs on a AMD K6 450 MHz machine with 256 megabytes of main memory. The two machines are connected directly by an 100-megabit fast Ethernet.

We employ the software Httperf to read a log of 100,256 queries submitted to TodoBR and to generate workload to the various server implementations at controlled rates. It measures the performance of the server from a client perspective, reporting, among other information, the average response time for the client to receive an answer, the throughput of the server, and occasional error rates.

The overall amount of server main memory used for the various cache implementations was set to 270 megabytes, based on the results presented in Section 5. In the two-level cache the memory was divided into two partitions: 20 megabytes for caching query results and 250 megabytes for caching inverted lists. A cache of 270 megabytes shows to be enough to achieve good performance in all cache schemes studied in this work and accounts for only 6.5% of the overall index size of TodoBR.

Implementations	**Processed Queries**	**Fetched Pages**
No Cache	100,365	5,509,684
Cache of Inverted List	110,296	446,269
Cache of Query Results	39,098	1,892,377
Two-Level Cache	49,128	456,275

Table1

Table 1 shows the counts for submitted queries and inverted list pages retrieved from disk, as an indication of CPU and disk demands for the four implementations. We can observe that caching query results reduces significantly (up to 62%) the number of queries that need to be processed.

On the other hand, caching inverted lists reduces the number of page reads by an order of magnitude. The two-level cache shows to be a good compromise in terms of performance, since it gets close to the best results, that is, the number of queries processed increases by only 21%, and the number of pages retrieved increases by only 3%.

At low request rates, the best performance was achieved by the cache of query results, which presents the lowest processing costs, closely followed by the two-level implementation, while the cache of inverted lists gives response times close to the implementation with no cache. This result is explained by the overhead associated with handling inverted lists and the gains inherent to the file system cache provided by the Linux operating system, which reduces the time to read a disk page.

At higher request rates the disk throughput saturates and the cache of inverted lists effectively improves the engine performance when compared to the implementation with no cache. The differences in the amount of disk operations also explain the better scalability of the two-level cache. As shown in Table 1, the two-level cache presented a miss ratio in terms of query results close to the miss ratio of the cache of query results. On the other hand, the total number of disk reads in the two-level cache was only 20% of the total number of reads performed when caching only query results.

An immediate consequence of the better performance provided by the two-level cache is a better overall throughput. The maximum throughput obtained by the two-level cache is 64 queries per second, while the maximum for the system with no cache was 22 queries per second. For the cache of inverted lists, the maximum throughput was 42 queries per second. For the cache of query results, the maximum throughput was 47 queries per second.

6. CONCLUSIONS

In this paper, we have proposed and evaluate experimentally a new multi-level caching architecture and scheme for web search engines that can improve query throughput and improve search engine scalability without modifying the ranking of query results. We have implemented and evaluated three different caching schemes on the search engine TodoBR, and compared the performance of these implementations to the original engine with no cache. The experiments show that the two-level cache provides the maximum throughput among all implementations, and that it is superior to the implementation with no cache by a factor of three. Furthermore, the throughput of the two-level cache is up to 52% higher than the implementation using just inverted lists and up to 36% higher than the cache of query results. The analysis of the TodoBR logs indicates that the miss ratios of both caches tend to decrease as we consider larger request streams. We are also interested in studying the impact of caching in search engines which are based on other ranking algorithms, such as ranking based on link analysis. The changes in the ranking algorithm can affect the cache system because the access pattern for the inverted lists may change and extra information may have to be retrieved from other index structures apart from the inverted lists. To our knowledge there is no published work on how to apply pruning to such types of ranking functions, which are not based on a simple combination of the scores for different terms.

REFERENCES

[1]. J. Zhang, X. Long, and T. Suel. Performance of Compressed Inverted List Caching in Search Engines. In WWW, 2008.

[2]. R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The Impact of Caching on Search Engines. In SIGIR, 2007.

[3]. A. Ntoulas and J. Cho. Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee. In SIGIR, 2007.

[4]. Y. Tsegay, A. Turpin, and J. Zobel. Dynamic Index Pruning for Effective Caching. In CIKM, 2007.

[5]. N. Laoutaris, S. Syntila, and I. Stavrakakis. Meta algorithms for hierarchical web caches. In IEEE International Performance Computing and Communications Conference (IEEE IPCCC), Phoenix, Arizona, April 2004.

[6]. K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In First Latin American Web Congress, pages 132–143, 2003.

[7]. T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. ODISSEA: A peer-to-peer architecture for scalable web search and information retrieval. In International Workshop on the Web and Databases (WebDB), June 2003.

[8]. K. Risvik and R. Michelsen. Search engines and web dynamics. Computer Networks, 39:289–302, 2002.

[9]. Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In IEEE Infocom 2002, pages 1238–1247, 2002.

[10]. K. Risvik and R. Michelsen. Search engines and web dynamics. Compute Networks, 39:289–302, 2002.

[11]. C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In Proc. of the 9th String Processing and Information Retrieval Symposium (SPIRE), Sept. 2002.

[12]. 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.

[13]. G. Navarro, E. S. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. Information Retrieval, 3(1):49-77, 2000.

[14]. Z. Lu and K. S. McKinley. Partial collection replication versus caching for information retrieval systems. In Proc. of the 23rd Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, pages 248-255, 2000.

[15]. I. Silva, B. Ribeiro-Neto, P. Calado, E. S. Moura, and N. Ziviani. Link-based and content based evidential information in a belief network model. In Proc. 23rd Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, pages 96-103, July 2000.

[16]. E. P. Markatos. On caching search engine results. In Proc. of the 5th Int. Web Caching and Content Delivery Workshop, May 2000.

[17]. N. Ziviani, E. S. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next generation text retrieval systems. IEEE Computer, 33(11):37-44, 2000.