

On Caching Search Engine Results

Evangelos P. Markatos
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385
Heraklio, Crete, GR-711-10 GREECE
tel: +30 81 391 655, fax: +30 81 391 661
markatos@csi.forth.gr

Technical Report 241

Abstract

In this paper we explore the problem of *Caching of Search Engine Query Results* in order to reduce the computing and I/O requirements needed to support the functionality of a search engine of the world-wide web.

Based on traces from search engines we show that there is significant locality in the queries asked, that is, 20-30% of the queries have been previously submitted by the same or a different user. Using trace-driven simulation we show that medium-sized caches can hold most of the frequently-submitted queries. Finally, we propose and evaluate a new cache replacement algorithm named LRU-2S, that takes into account both the frequency and the recency of access to a page when making a replacement decision.

1 Introduction

Search Engines have become a popular way of finding information on the web. People that are interested in a particular topic, send a query about the topic to a search engine and based on the returned URLs they start their information finding process. A recent survey by CommerceNet/Nielsen Media suggests that more than 70% of users reach web sites via a Search Engine. As the web becomes larger, individual sites become more difficult to find, and thus the percentage of users that reach web sites via a Search Engine will probably increase. Besides people, computers may also routinely access search engines. For example, web agents (robots) may periodically inquire search engines about *new* information on specific subjects [12].

In this paper we explore the problem of *Caching of Search Engine Query Results* in order to reduce the computing and I/O requirements needed to support the functionality of a search engine. Each query performs a database search to find and return a number of URLs (and maybe a summary for each URL) that match the given query. Depending on the complexity of the query the search may make several accesses to data/metadata (that may reside in secondary storage) and may consume significant computing resources. If a query is popular, then caching its results may result in significant performance improvements, since the query results will be calculated only once: when the query is submitted for the first time.

One might argue that caching is not desirable since caching query results for a long time may result in stale data which will be useless to the person that submitted the query. Fortunately, caching query results for a few hours is enough to achieve most of the benefits of caching. Caching query results even for several days may not significantly contribute to stale data, since search engines routinely return URLs that are as old as a few months old. This is because the spiders of the most popular search engines visit the sites they index every month, or so. This implies that the results returned by a search engine query (independent of whether caching is employed) may include data that are a month old. Thus, caching query results for a few hours (or even days) will not influence whether the returned data are stale.

In this paper we show that caching is a viable way to improve performance of search engines. The contributions of the paper are:

- We study the traces of a popular Search Engine (Excite) and show that there is a significant locality in the queries asked, that is, 20-30% of the queries have been previously submitted by the same or a different user.
- Using trace-driven simulation we show that medium-sized caches are enough to hold the results of most of the repeatedly submitted queries.
- We propose a new cache replacement algorithm named *two-stage* LRU (LRU-2S) that combines both the frequency and the recency of access to a page when making a replacement decision. We show that LRU-2S performs comparable to (and usually better than) previously proposed algorithms for database and file system buffer caching (like LRU/K, and FBR), while at the same time retaining the (constant time) efficiency of the simple LRU cache replacement algorithm.

2 Previous Work

Caching documents to reduce access latency is being extensively used on the web. Most web browsers cache documents in the client's main memory or in some local disk. Although this is the most widely used form of web caching, it is the least effective, since it rarely results in large hit rates [1]. To improve cache hit rates, caching proxies are used. Proxies employ large caches which they use to serve a stream of requests coming from a large number of users (clients). Since even large caches may eventually fill up, cache replacement policies have been the subject of intensive recent research. One of the first cache replacement policies considered was LRU (Least Recently Used) and its variations. LRU is based on the heuristic that the documents not used recently will probably not be requested in the near future. After studying the access patterns of several web clients, it was realized that small documents were requested more often than large documents. Thus, LRU was extended with heuristics that favor caching of small documents, by precluding caching of large documents [1, 10], or by eagerly removing large documents [1, 14, 21]. Given that clients experience different latency to different servers cache replacement policies may also take network latency into account, in order to avoid replacing documents that may take a lot of time to download [16, 22]. Some policies aggregate all the above factors (access recency, size, latency) into a "weight" or "value" and try to keep in the cache the documents with the largest values [4, 8]. Some approaches propose the development of hierarchical caches that cooperate in order to provide a higher hit rate capitalizing on a larger number of clients and a larger cumulative amount of storage [5, 9]. Finally, some caches employ intelligent *prefetching* methods to improve the hit rate even further [2, 3, 6, 11, 20, 19].

Spink *et al.* have analyzed the transaction logs of queries posed by users of *Excite*, a major Internet Search Engine [7, 17]. They have focused on *how* do users search the web, and on *what* do they search on the web. They present a large number of findings. Among the most interesting ones are that users (in average) ask short queries and that they view 2-3 pages of the results.

There also exist various on-line sources where one can find useful statistics about search patterns on the web. For example, a user may find the most popular keywords used in some search engines or how many times a given query was asked in some particular search engine. For more information see <http://www.searchengineguide.org>

To the best of our knowledge there has been no previous study of the existence of locality in search engine queries, and of evaluating the performance of caching their results. However, there exists a significant amount of research in buffer replacement in file systems and databases. The most widely known replacement policy, the LRU, replaces the buffer that has been Least Recently Used. Although LRU favors recently used pages it does not significantly favor frequently used pages. Robinson and Devarakonda proposed FBR, an algorithm that takes into account both the recency and frequency of access to a page [15]. The algorithm can be summarized as follows: "(1) the blocks in the cache are totally ordered in the same fashion as a cache using LRU replacement; (2) the cache is divided into three parts, a *new* section containing the most-recently used blocks, an *old* section containing the least-recently used blocks, and a *middle* section between the new and the old sections; (3) reference counts are only incremented for blocks that are not in the new section; and (4) on a miss, the block with the smallest

reference count in the old section is replaced (with the least-recently-used such block selected if there is more than one).

O’Neil *et al.* proposed the LRU/2 replacement algorithm: LRU/2 replaces the page whose penultimate (second-to-last) access is least recent among all penultimate accesses [13]. They also proposed LRU/K, a generalization of LRU/2: it replaces the page whose K-to-last access is least recent among all such accesses.¹

Johnson and Shasha proposed the 2Q buffer management replacement algorithm that achieves similar performance to LRU/K but is significantly faster to execute [18]. 2Q works as follows: (1) the cache is divided into two parts, a *Am* section containing the frequently used blocks, an *A1* section; (2) on a hit, the buffer is moved to the head of the section it belongs to; (3) on a miss, the buffer is put in the head of *A1*; (4) when the algorithm runs out of space it pages out the tail of *A1*, unless *A1* is very small, in which case it pages out the tail of *Am*.

All the above three cache replacement policies take into account both the frequency and the recency of access when making a replacement decision. LRU/K usually has the best hit rates but is expensive to compute (it takes logarithmic time per decision). On the other hand, 2Q has somewhat lower hit rates but (like FBR) computes much faster - in constant time per decision. Our two-stage LRU algorithm combines the best of both sides: it computes in constant time, and usually achieves the best (or very close to the best) hit rate.

3 Experiments

3.1 The Experimental Environment

3.1.1 The Traces

We use traces from the Excite search engine (www.excite.com) The traces contain about one million searches. Of them 927,010 are queries and the rest are requests for “similar” documents². Each query is actually a request for a particular page of the results of some query. Thus, when a user requests the first page of results of a given Query this represents one trace record in the log file. When the same user later requests the second page of the results of the same Query, this request is written as a separate trace record in the log file. Thus, from now on, when we say query, we actually mean “a particular page of the the results of a given query”. In this sense all (most) queries have the same size - about a page of results long (which roughly corresponds to 4 Kbytes).

3.2 Locality of Reference

In our first experiment we set out to find if there exists any locality of reference among the queries submitted, that is if queries are submitted more than once. Figure 1 plots the number of accesses for the 1,000 most popular queries. Our results suggest that some queries are very popular: the most popular query is submitted 2,219 times. Moreover, there exists a large number of queries that are accessed several times and are excellent candidate for caching: even the 1,000th most popular query is submitted as many as 27 times.

Although Figure 1 suggests that several queries are being submitted repeatedly, it does not quantify the temporal locality of the queries submitted, that is, is the same query repeatedly submitted within a small time interval? To quantify the temporal locality of the queries submitted we measured the time between successive submissions of the same query. The time is measured in queries intervened between the two successive submissions. The results were rounded to the nearest hundred and plotted in Figure 2. We see that in 1639 instances the time between successive submissions of the same query was less than 100. In other words, a cache size that can hold the results of only 100 queries is enough to result in 1639

¹The above description of LRU/2 is oversimplified. If LRU/2 worked as stated it would soon degenerate to MRU, and would replace the block that has just brought in because eventually all blocks in the cache would have a positive penultimate access time, while newly accessed blocks would have a $-\infty$ penultimate access time, leading to their replacement. To avoid such degenerate behavior, O’Neil *et al.* suggest that blocks once brought into the memory buffer should stay there for a number of seconds. In our implementation of LRU/2, we keep all blocks in the cache according to their LRU access time and replace according to LRU/2 only from the last two thirds of the buffer. Thus, new blocks are given the chance to grow old and have a positive penultimate access time.

²Excite gives users the ability to find documents that are “similar” to a URL returned.

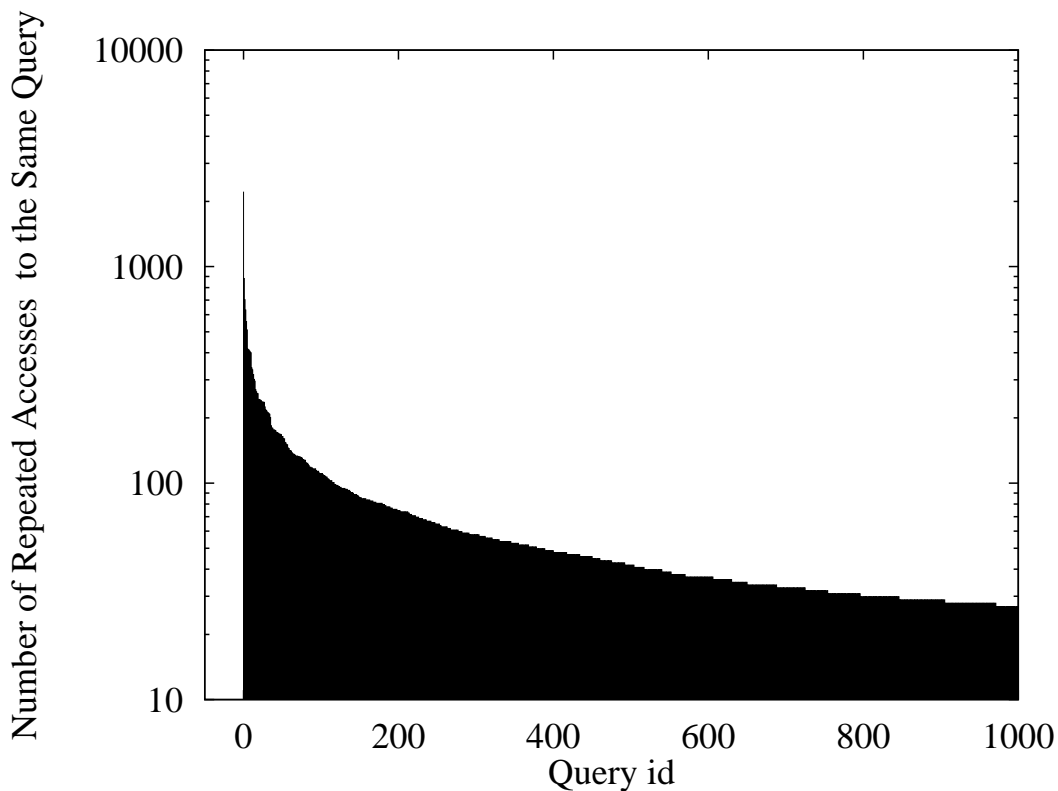


Figure 1: **Number of accesses for the 1,000 most popular queries.** The graph displays the number of requests for the 1,000 most popular queries of the trace studied. We see that the most popular query is submitted 2,219 times, while the 1,000th most popular query is submitted 27 times.

hits. Figure 2 also shows that there exist a significant number of queries whose successive re-submissions are quite apart from each other. This fact implies that we may need sophisticated cache replacement algorithms to make sure that the results of frequently submitted queries stay in the cache and are not replaced by the results of queries that are infrequently resubmitted.

3.3 Caching

Since our preliminary results suggest that there exists temporal locality in the queries submitted, it seems worthwhile to save the results of some (all?) queries in a cache, so that subsequent submissions of the same queries should be serviced from the cache. Caching query results may save

- execution time, because no query will be executed
- temporary storage space, because no intermediate results of the query will be generated
- I/O time, because no access to the database will be needed

The cache for the query results may be in main memory and/or in secondary storage, depending on the cache size.

3.3.1 Two-stage LRU

We propose a new cache replacement algorithm named two-stage LRU (LRU-2S) that works as follows: (1) the blocks in the cache are totally ordered in the same fashion as a cache using LRU replacement; (2) the cache is divided into two parts, a *primary* section and an *secondary* one; (3) on a miss, the last block of the secondary section is replaced, and the new block is put on the head of the secondary section; (4) on a hit, the block is moved to the head of the primary section - if the block was in the secondary section, then the last block of the primary becomes the first of the secondary section. Figure 3 presents a few snapshots of the state of the queue kept by LRU-2S.

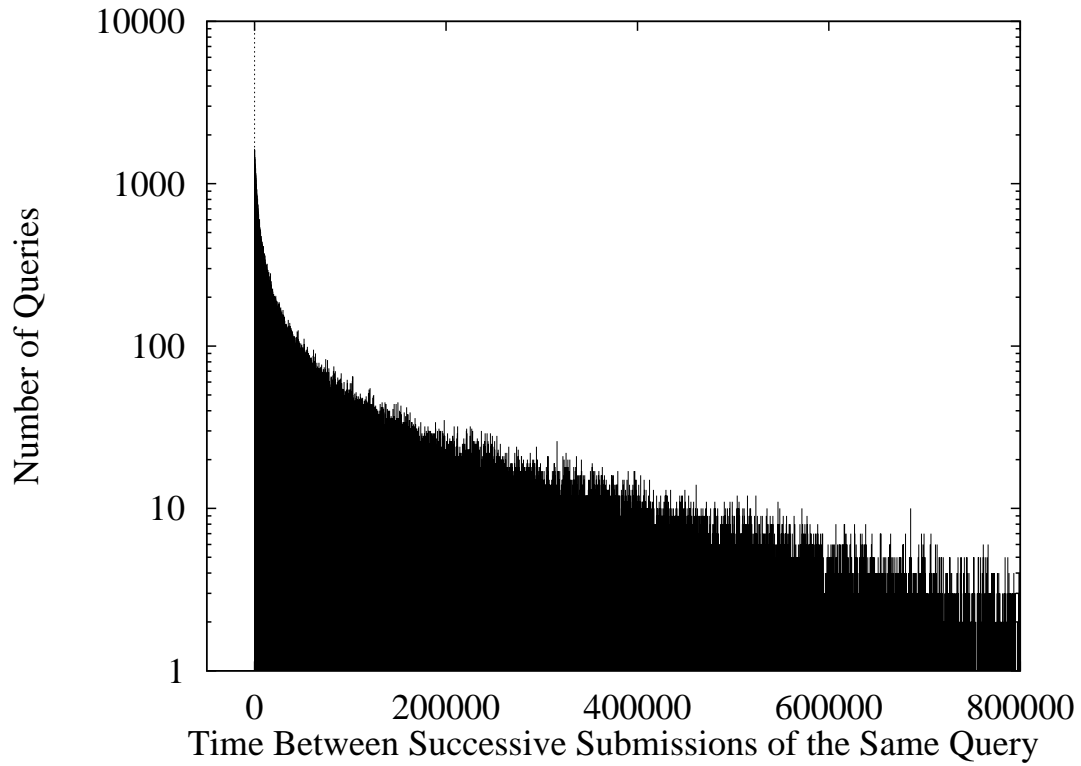


Figure 2: **Distances between submissions of the same query.** The graph displays the histogram of times between submissions of the same query by some user. We see that there is a large number of queries (1639) whose time between two successive submissions is less than 100 (other) queries. We also see that the distribution has a heavy tail. This implies that there exists a large percentage of queries, whose successive submissions are quite apart from each other.

Initial state of the Queue:			
primary		secondary	
4	3	2	1
After accessing 1:			
1	4	3	2
After accessing 4:			
4	1	3	2
After accessing 5:			
4	1	5	3
After accessing 5:			
5	4	1	3
After accessing 6:			
5	4	6	1

Figure 3: **Example of LRU-2S.** LRU-2S keeps all blocks in a queue. The first part of the queue keeps the frequently accessed blocks while the second part keeps the infrequently accessed blocks. Accesses that hit in the cache move the block to the head of the primary section, while misses put the newly accessed block to the head of the secondary queue. In this example, the queue initially contains blocks 4, 3, 2, 1. After accessing block 1, it is moved to the head of the primary queue, and all other blocks move one position back (they “scroll down”), which effectively moves block 3 from the primary queue to the secondary queue.

LRU-2S is in several ways similar to previous algorithms. Much like FBR, LRU-2S partitions the cache into sections that separate frequently used blocks from infrequently used ones. However, LRU-2S employs no counters (as FBR does) and employs simpler (faster) data structures. LRU-2S is also similar to 2Q in that they both partition the cache into 2 sections. However, 2Q replaces buffers from both sections, while LRU-2S replaces buffers only from the secondary section. Also, 2Q may keep a block for a long time in the second section even if it is accessed frequently, while LRU-2S is always eager to promote a block to the first section. Finally, 2Q keeps information about blocks that are not present in the cache, while LRU-2S does not.

3.3.2 Cache Replacement

No matter what the size of the cache we use, it is bound to fill up with query results after a while. When that happens we should replace some results from the cache. The cache replacement algorithm may play an important role in the exact performance of the cache.

We have studied the following cache replacement policies:

- LRU
- FBR
- LRU/2
- LRU-2S

The performance metric we will use is the *hit rate*, that is the percentage of queries whose results can be found on the cache. Figure 4 plots the hit rate as a function of cache size. We see that the performance (hit rate) of all algorithms increases with cache size. For caches larger than 1 Gbytes all algorithms perform very close to the best ³. For very small caches (<100 Mbytes) FBR and LRU-2S have the best performance followed by LRU/2 and LRU. For mid-size caches, LRU/2, LRU-2S and FBR have similar performance followed by LRU. For large caches (> 1 Gbyte) all algorithms have similar performance.

Figure 5 presents the hit rates achieved by FBR, LRU-2S, and LRU/2 alone in logscale in order to focus on their differences. We see that for very small cache sizes FBR performs better than LRU/2.

³The cache size that would be required to hold the results of *all* queries submitted is 2.5 Gbytes.

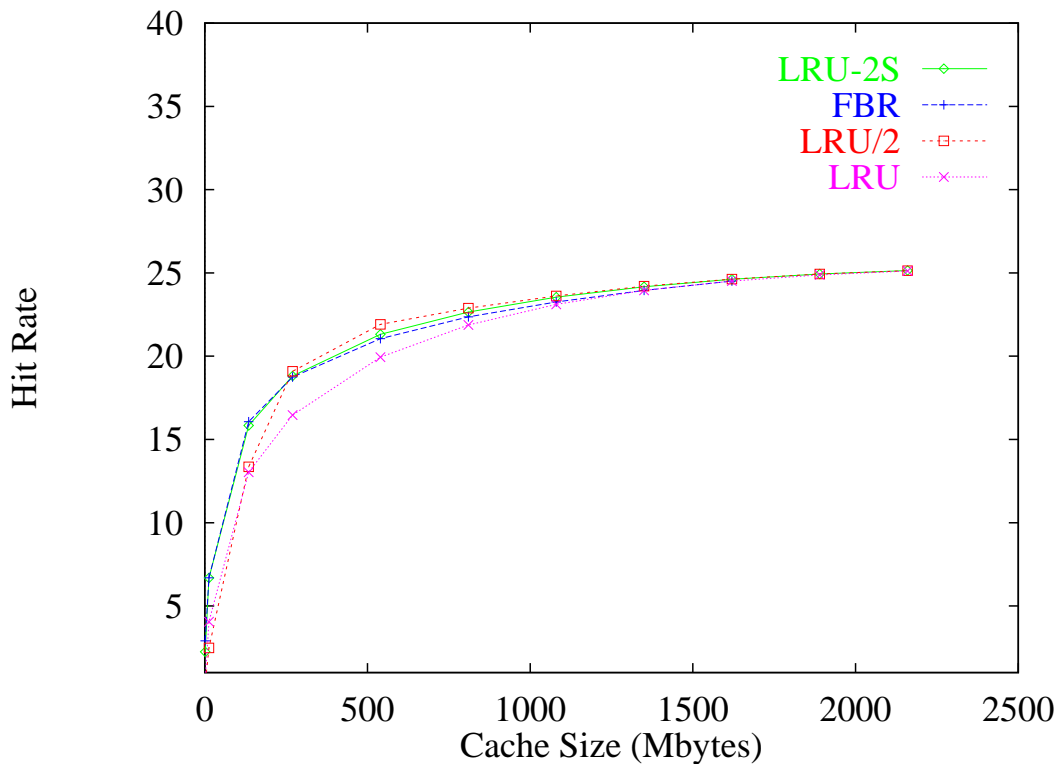


Figure 4: **Hit Rate as a function of the Cache Size**

For large caches LRU/2 performs a little better than FBR. However, in all cases, LRU-2S performs very close to the best of FBR and LRU/2.

Figure 4 suggests that the maximum hit rate that can be achieved for large caches is around 25%. Unfortunately, during our experiments a significant percentage of the time was spent to warm up the caches, during which time hit rates were very low. To see what is the hit rate of the system with warm caches we measured the hit rate for each interval of successive 50,000 queries for the LRU and LRU-2S replacement policies. Figure 6 plots this hit rate as a function of time for cache size equal to 1.6 Gbytes (60%) of the maximum cache size. We see that at the beginning the hit rate is low, but increases with time. Halfway through the simulation (at 450,000 submissions) it has reached a 25%. From that point on it continues to increase (albeit slowly) and stabilizes in the area of 29%. This experiment suggests that once the caches are warmed up, their effectiveness may reach close to 30%. To put it simply, (roughly) one out of three queries submitted will be served from the cache.

3.3.3 Performance Tuning

In this section we will explore how sensitive are LRU-2S and FBR to their parameters, and which ranges of parameters are appropriate for each caching configuration. Figure 7 plots the hit rate of LRU-2S for various percentages of the primary region. The figure plots results for small-sized and medium-sized caches, since all cache replacement policies do not display noticeable performance differences. The first thing we notice is that there are no significant performance differences among the various versions of LRU-2S studied, which implies that although good choices of parameters may improve performance, bad choices do not result in significant performance decrease.

For small cache sizes (13 Mbytes) the hit rate increases with the size of the primary region. This is expected because the primary region keeps the frequently accessed queries which amount for the highest percentage of hits. For medium size caches performance initially increases with the size of the primary region, and then decreases. In all cases studied (135-811 Mbytes) fixing the primary region to the 30% of the total cache size resulted in best (or very close to best) performance. Fortunately, deviations from that percentage do not noticeably hurt performance. For example, if the primary region is set to more than twice as large (70%), the performance is practically the same.

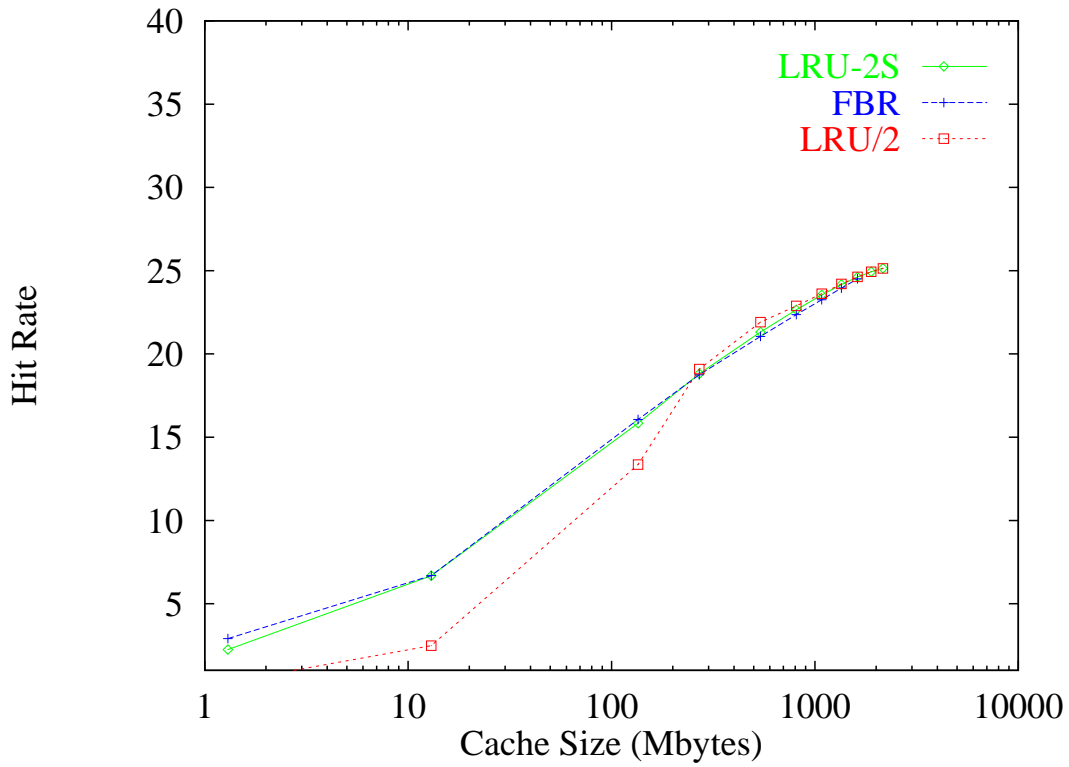


Figure 5: Hit Rate of FBR, LRU-2S, and LRU/2 as a function of the Cache Size

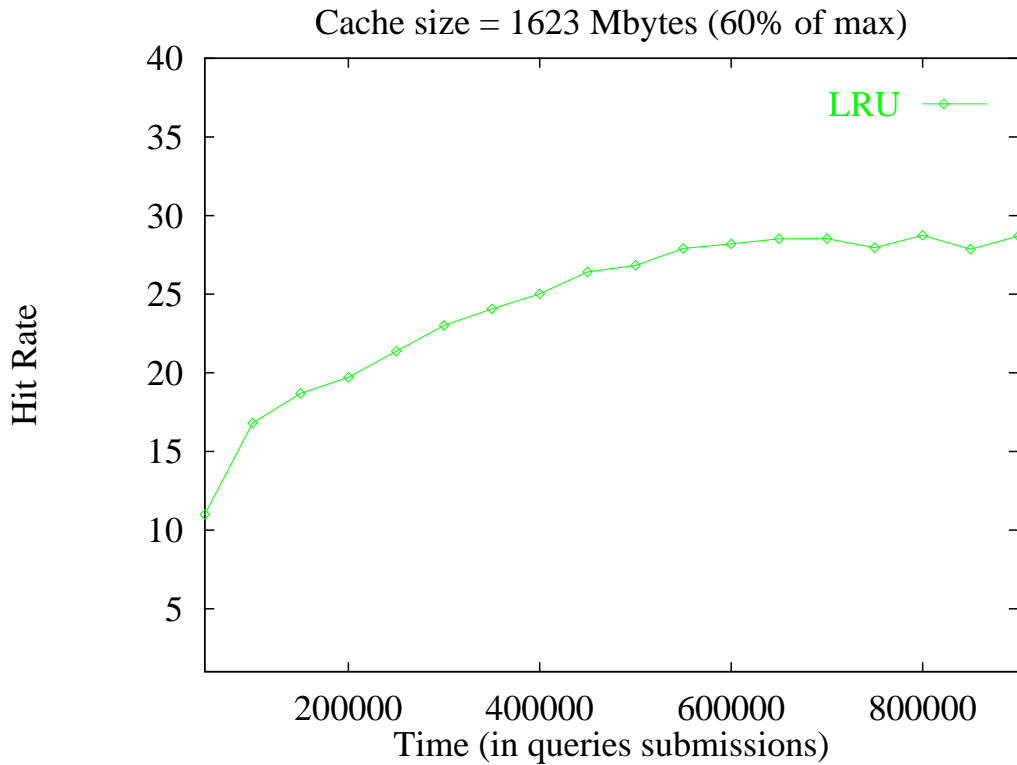


Figure 6: Hit Rate as a function of Time. We see that all policies start with a low hit rate. When they warm up the caches, hit rates may reach up to 30%.

Figure 8 plots the hit rate of LRU-2S as a function of the size of the primary section (much like Figure 7 but for a wider range of parameters). We see that the performance of LRU-2S is almost flat through the entire range of parameters. We see some noticeable reduction in performance only for extreme values (e.g. higher than 0.95). Overall, the performance of LRU-2S is not sensitive to reasonable changes in the division of the cache size into the primary and secondary sections.

Figure 9 plots the hit rate of FBR for various percentages of the new, middle and old sections. It can be easily seen that FBR is robust under various configurations, and thus (similarly to LRU-2S) a bad choice of parameters does not significantly affect its performance. We also see that for small cache sizes configurations with large old sections tend to perform better since it is there where they keep the frequently (but not recently) accessed documents which increase the hit rate. For medium sized caches, the performance is rather stable for any value of the parameters chosen.

3.4 Execution Efficiency

It is clear from our experiments that all the frequency-conscious algorithms achieve similar hit rates for the range of cache sizes where it is reasonable to employ caching. Thus, choosing the most appropriate for each case may depend on other factors including efficiency of execution: i.e. how fast the algorithm executes when a replacement decision has to be made.

LRU/2 replaces the page whose penultimate access time is the oldest of the penultimate access times of all pages. Finding this page requires the maintenance of a priority queue. Each time a page is accessed, its penultimate access time changes and its position in the priority queue needs to change. Also, when a replacement needs to be made, the page with the oldest penultimate access is removed from the priority queue and the queue needs to be re-organized. Both priority queue operations take $O(\log N)$ time where N is the number of blocks in the cache (which can thousands or even millions).

FBR can be also implemented in $O(\log N)$ (worst case) access time using a priority queue to keep the sorted values of reference counters. It is possible, however, to implement FBR in constant (average case) access time using several FIFO queues, one for each value of the access count. When a replacement needs to be made, the queues are scanned (from the lower to the higher values). The scanning stops when the first queue that holds an *old* LRU block is found. If the queues that correspond to low access counts have old blocks then the algorithm executes fast - probably in constant time.

LRU-2S can easily be implemented in constant time using only a single LRU queue with three pointers: `head`, `tail` and `middle`. The `head` points to the head of the primary queue, the `middle` is the tail of the primary queue and the head of the secondary queue, and the `tail` is the tail of the secondary queue. On a block hit, the block is moved to the front of the primary queue (in constant time). On a block miss, the block is placed at the front on the secondary queue (in constant time). When a replacement needs to be made, the last block of the secondary queue is evicted (in constant time). The following table shows the data structures needed and the time complexity of the various replacement algorithms:

algorithm	data structure	time complexity
LRU/2	priority queue	$O(\log N)$
FBR (simple)	priority queue	$O(\log N)$
FBR (sophisticated)	several FIFO queues	$O(1)$
LRU-2S	1 FIFO queue	$O(1)$

Summarizing, we believe that LRU-2S is the simplest algorithm to implement, the fastest to execute, while at the same time achieving performance that is comparable to (and sometimes better than) the best of FBR and LRU/2.

4 Conclusions

In this paper we explore the problem of *Caching of Search Engine Query Results* in order to reduce the computing and I/O requirements needed to support the functionality of a search engine. We use query traces from the EXCITE search engine to find the locality that may exist in the queries submitted and evaluate the performance of various cache replacement algorithms. Based on our trace-driven simulations we conclude:

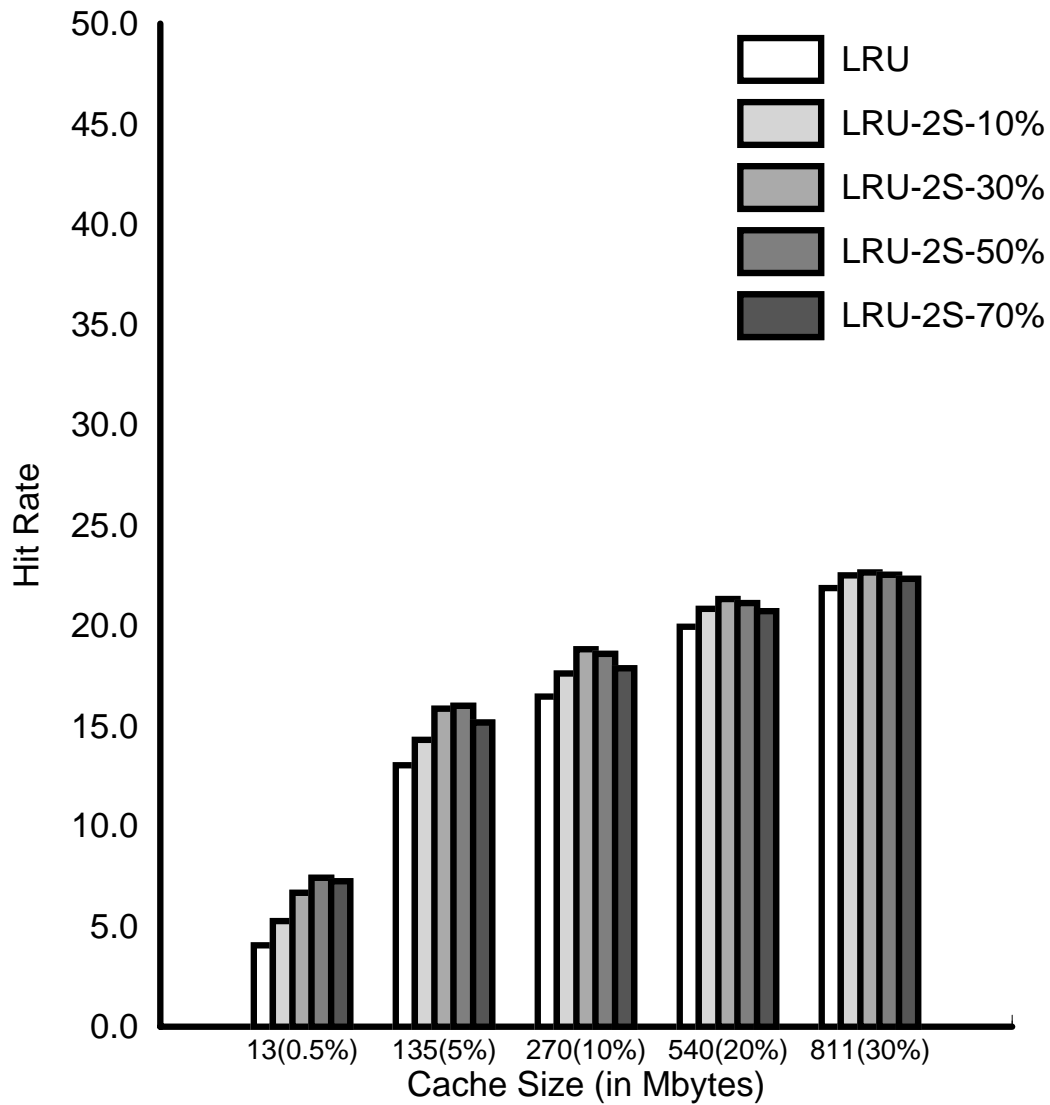


Figure 7: **Hit Rate for various configurations of LRU-2S.** LRU-2S-10% means that 10% of the cache size holds the primary section and the rest 90% holds the secondary section.

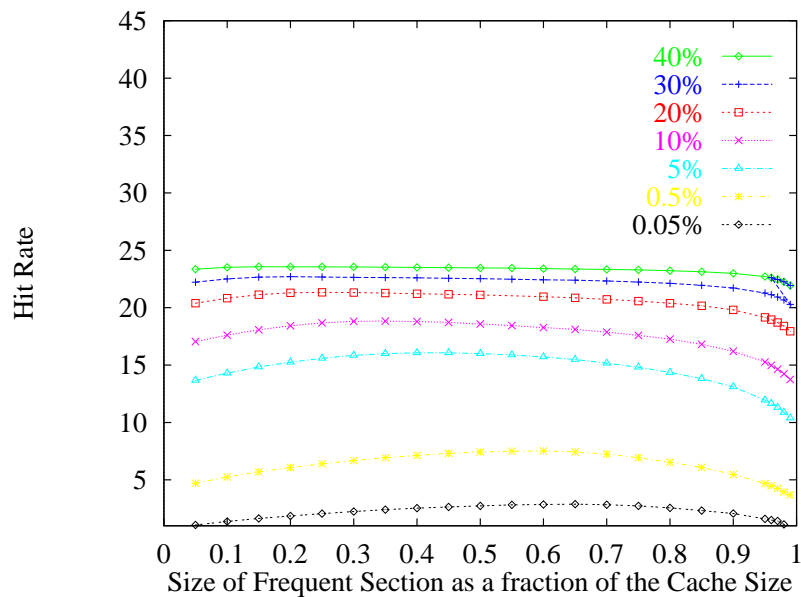


Figure 8: **Hit Rate for various configurations of LRU-2S.** Each line in the graph is for a different cache size. For example, the 10% line represents a cache size that is 10% of the max cache size needed. We see that the hit-rate of LRU-2S is rather stable, apart from extreme values of parameters (e.g. larger than 0.95 or smaller than 0.05).

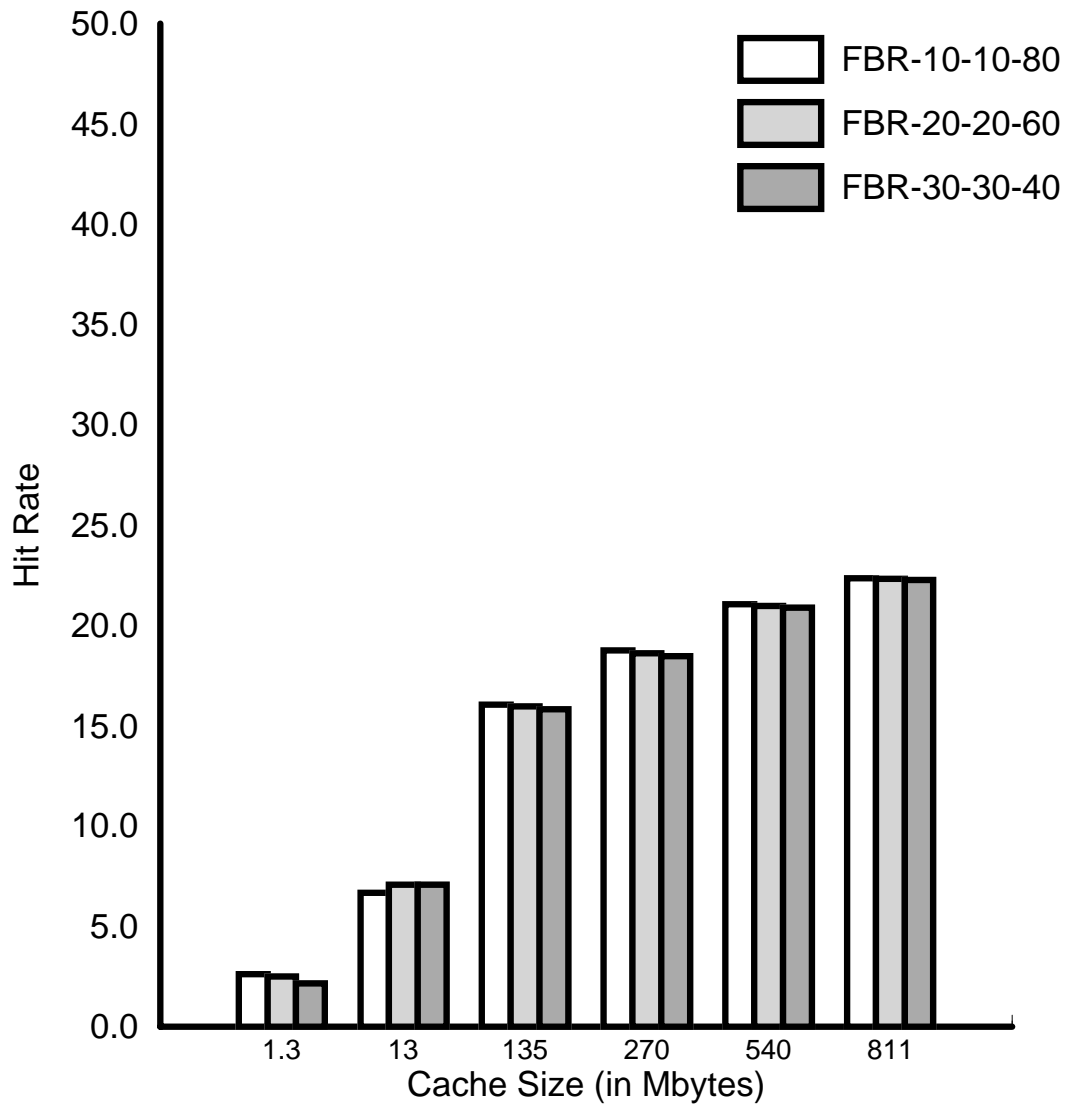


Figure 9: **Hit Rate for various configurations of FBR** FBR-10-10-80 means that 10% of the buffer is the new section, 10% is the middle section and 80% is the old section.

- *There exists a significant amount of locality in the queries submitted to popular web search engines.* Our experiments suggest that one out of three of the queries submitted has been previously submitted by the same or by another user.
- *Medium-sized caches (a few hundred Mbytes large) can easily exploit the locality found and serve a significant percentage of the submitted queries from the cache.* Our experiments suggest that medium sized-caches can result in hit rates ranging from 25% to 30% (for warm caches).
- *Effective Cache Replacement Policies should take into account both recency and frequency of access in their replacement decisions.* Our experimental results suggest that FBR, LRU/2, and LRU-2S always perform better than simple LRU which does not take frequency of access into account.
- *The proposed LRU-2S replacement policy has performance comparable to (and sometimes better than) previously proposed cache replacement policies while maintaining a low execution cost.* In all our experiments LRU-2S, LRU/2, and FBR had comparable performance. FBR was better than LRU/2 for small caches, while LRU/2 was a little better for large caches. In all cases, LRU-2S was very close to the best of LRU/2 and FBR.

Acknowledgments

This work was supported in part by PENED project 2041 2270/1-2-95. We deeply appreciate this financial support.

Some of the experiments were run at the Center for High-Performance Computing of the University of Crete (<http://www.csd.ucl.ac.uk/hpcc/>). Pei Cao provided the initial version of the simulator used in this paper. Amanda Spink and Jim Larsen pointed out the EXCITE traces and helped us with several questions. We thank all of them.

References

- [1] M. Abrams, C.R.Standridge, G. Abdulla, S. Williams, and E.A. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of the Fourth International WWW Conference*, 1995.
- [2] Azer Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems. In *Proceedings of ICDE'96: The 1996 International Conference on Data Engineering*, March 1996.
- [3] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *proceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management*, November 1995.
- [4] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [5] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, March 1995.
- [6] J. Gwertzman and M. Seltzer. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.
- [7] B.J. Jansen, A. Spink, J. Bateman, and T. Saracevic. Searchers, the subjects they search, and sufficiency: A study of a large sample of EXCITE searches. In *Proceedings of Webnet 98*, 1998.
- [8] P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement Policies for a Proxy Cache, 1998. <http://www.iet.unipi.it/luigi/research.html>.
- [9] Radhika Malpani, Jacob Lorch, and David Berge. Making World Wide Web Caching Servers cooperate. In *Proceedings of the Fourth International WWW Conference*, 1995.

- [10] E.P. Markatos. Main Memory Caching of Web Documents. *Computer Networks and ISDN Systems*, 28(7-11):893–906, 1996.
- [11] E.P. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *Proceedings of the INET 98 Conference*, 1998.
- [12] Evangelos P. Markatos, Christina Tziviskou, and Athanasios Papathanasiou. Effective Resource Discovery on the World Wide Web. In *Proceedings of Webnet 98*, pages 611–616, 1998.
- [13] E.J. O’Neil, P.E. O’Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. of the 1993 ACM SIGMOD Conf. on Management of Data*, pages 297–306, 1993.
- [14] J.E. Pitkow and M. Recker. A Simple, Yet Robust Caching Algorithm Based on Dynamic Access Patterns. In *Proceedings of the Second International WWW Conference*, 1994.
- [15] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proc. of the 1990 ACM SIGMETRICS Conference*, pages 134–142, 1990.
- [16] P. Scheuearmann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *6th International World Wide Web Conference*, 1997.
- [17] A. Spink, B.J. Jansen, and J. Bateman. Users’ searching behavior on the EXCITE web search engine. In *Proceedings of Webnet 98*, 1998.
- [18] Dennis Shasha Theodore Johnson. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. 1994 Very Large Data Bases*, pages 439–450, 1994.
- [19] Joe Touch. Defining High Speed Protocols : Five Challenges and an Example That Survives the Challenges. *IEEE JSAC*, 13(5):828–835, June 1995.
- [20] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast World-Wide Web Browsing Over Low-Bandwidth Links, 1996. <http://ccnga.uwaterloo.ca/sbwachsb/paper.html>.
- [21] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of the ACM SIGCOMM 96*, 1996.
- [22] Roland P. Wooster and Marc Abrams. Proxy Caching that Estimates Page Load Delays. In *6th International World Wide Web Conference*, 1997.