A Highly Scalable Parallel Caching System for Web Search Engine Results

T. Fagni[1], R. Perego[1], and F. Silvestri[1,2]

[1] Istituto ISTI, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. This paper discusses the design and implementation of SDC, a new caching strategy aimed to efficiently exploit the locality present in the stream of queries submitted to a Web Search Engine. SDC stores the results of the most frequently submitted queries in a *fixed-size read-only* portion of the cache, while the queries that cannot be satisfied by the static portion compete for the remaining entries of the cache according to a given cache replacement policy. We experimentally demonstrated the superiority of SDC over purely static and dynamic policies by measuring the hit-ratio achieved on two large query logs by varying cache parameters and the replacement policy used. Finally, we propose an implementation optimized for concurrent accesses, and we accurately evaluate its scalability.

1 Introduction

Due to the high locality present in the stream of queries processed by a Web Search Engine (WSE), caching the results of the queries submitted by users is a very effective technique to increase the throughput. WSE results caching, similarly to Web page caching, can occur at several places, e.g. on the client side, on a proxy, or on the server side. Caching on either the client or the proxy has the advantage of saving network bandwidth. Caching on the server side, on the other hand, has the effect of saving I/O and computational resources used by the WSE to compute the page of relevant results to be returned to a user. One of the issues related to server-side caching is the limited resources usually available on the server, in particular the RAM memory used to store the cache entries. However, the architecture of a scalable, large-scale WSE is very complex and includes several machines which take care of the various subtasks involved in the processing of user queries. The distributed architecture of a large-scale WSE [7, 1] is composed by a farm of identical machines running multiple WSE *Core* modules, each of which is responsible for searching the index relative to one specific sub-collection of documents. In front of these searcher machines we have an additional machine hosting a *Mediator*. This module has the task of scheduling the queries to the various searchers, and of collecting the results returned back. Note that multi-threading is exploited extensively by all these modules in order to process concurrently distinct queries. Within this architecture the RAM memory is a very precious resource for the machines that

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 347–354, 2004.

host the WSE Core, which perform well only if the mostly accessed sections of their huge indexes can be buffered into the main memory. Conversely, the RAM memory is a less critical resource for the machine that hosts the Mediator. This machine can thus be considered as an ideal candidate to host a server-side cache. The performance improvement which may derive from the exploitation of query results caching at this level is remarkable. Queries resulting in cache-hits can be in fact promptly served thus enhancing WSE throughput, but also the queries whose results are not found in cache benefit substantially due to the lower load on the WSE and the consequent lower contention for the I/O, network and computational resources.

Beside the fact that several papers analyzed query logs to study the behavior of WSE users, only a few works propose effective techniques to exploit the locality present in the stream of user requests[5, 9, 4]. Markatos compared several caching policies on the basis of the hit-ratio obtained on a actual log of queries submitted to Excite [5]. The work by Saraiva *et al.* [9], discusses instead a two-level caching system which try to enhance the responsiveness of a hierarchically-structured search engine The first-level cache is similar to the one discussed in [5], while, the second-level cache is intended to store the posting lists of the keywords contained into the query strings. Finally, Lempel and Moran recently proposed *PDC* (Probabilistic Driven Caching), a new effective caching policy which associates a probability distribution to all the possible queries that can be submitted to a WSE. The distribution is built over the statistics computed on the previously submitted queries, and is used to compute a priority exploited to maintain an importance ordering among the entries of the cache.

In this paper, we are interested in studying the design and implementation of such a server-side cache of query results. Starting from the analysis of the content of two real query logs, we propose a novel replacement policy (called SDC- Static and Dynamic Cache) to adopt in the design of a fully associative cache of query results. According to SDC, the results of the most frequently accessed queries are maintained in a fixed size set of statically locked cache entries. This *Static Set* is rebuilt at fixed time intervals using statistical data coming from WSE usage data. When a query cannot be satisfied by the Static Set, it competes for the use of a *Dynamic Set* of cache entries. The management of the *Dynamic Set* can exploit, in principle, any replacement policy. In all the tests performed, the presence of a static portion of the cache resulted in large advantages both on the hit-ratio achieved and on the overall throughput of the caching system. We experimentally evaluated SDC by measuring the hit-ratio and the throughput achieved on actual query logs by varying the size of the cache, the percentage of cache entries of the Static Set, and the replacement policy used for managing the Dynamic Set. Differently from the other works, we also accurately assessed the scalability of our caching system with respect to the number of concurrent threads using it. The paper is organized as follows. Section 2 describes the query logs used, while in Section 3 we discuss our novel caching policy. Section 4 shows the results of the simulations performed on the different query logs. Finally, Section 5 presents some concluding remarks.

2 Analysis of the Query Logs

In order to evaluate the behavior of different caching strategies we used query logs from the Tiscali and Altavista search engines. In particular we used *Tiscali*, a trace of the queries submitted to the Tiscali WSE engine (www.janas.it) on April 2002, and *Altavista* a query log containing queries submitted to Altavista on the Summer of 2001[1]. Each record of a query log refers to a single query submitted to the WSE for requesting a *page* of results, where each page contains a fixed amount of URLs ordered according to a given rank. All query logs have been preliminarily cleaned by removing the useless fields. At the end of this preprocessing phase, each entry of a query log has the form *(keywords, page_no)*, where *keywords* corresponds to the list of words searched for, and *page_no* determines which page of results is requested. We further normalize the query log entries by removing those referring to requests of more than 10 results-per-page. After the cleaning, the *Altavista* query log contains $6,175,648$ queries of which $2,657,410$ are distinct. The *Tiscali* logs is instead composed of $3,278,211$ queries of which $1,538,934$ are distinct. Thus in both the logs about the 54% of the total number of queries are repeated more than once. The plots reported in Figure 1 assess the locality present in the query logs using a log-log scale. In particular Figure 1.(a) plots the number of occurrences within each log of the most popular queries, whose identifiers have been assigned in decreasing order of frequency. Note that, in all the two logs, more then $10,000$ different queries are repeated more than 10 times. Since the number of occurrences of a given query is a measure that might depend on the total number of records contained in the logs, to better highlight temporal locality present in the logs we also analyzed the time interval between successive submissions of the same query. The rationale is that if a query is repeatedly submitted within a small time interval, we can expect to be able to retrieve its results even from a cache of small size. Figure 1.(b) reports the results of this analysis. For each query log we plotted the cumulative number of resubmissions of the various queries as a function of the time interval (expressed as a distance measured in number of queries). Once more the results are encouraging: in the *Tiscali* log for more than $350,000$ times, the time interval between successive submissions of the same query is less than 100; in the *Altavista* log this temporal locality is slightly smaller than that in *Tiscali* but, again, for more than $150,000$ times the time interval is still less than 100.

3 The SDC Policy

SDC is a two-level policy which makes use of two different sets of cache entries. The first level contains the *Static Set* consisting in a set of statically locked entries filled with the most frequent queries appeared in the past. The Static Set is periodically refreshed. The second level contains the *Dynamic Set*. Basically, it is a set of entries managed by a classical replacement policy (i.e. *LRU*, *SLRU*,

[1] We are very grateful to Ideare S.p.A. and to Ronny Lempel for providing us with these query logs.

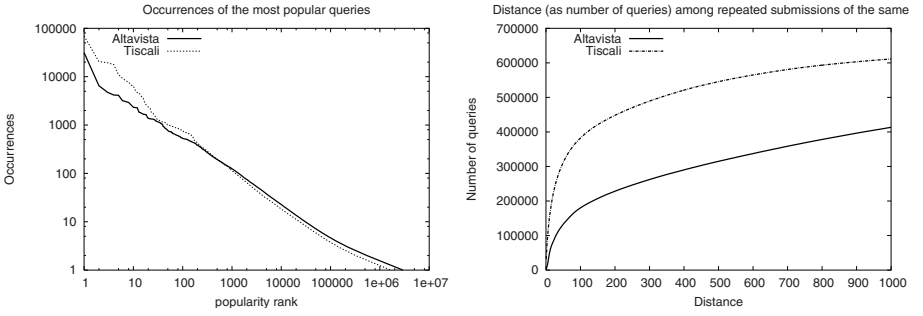

Fig. 1. Analysis of the locality present in the query logs.

etc.). The behavior of SDC in the presence of a query q is very simple. First it looks for q in the Static Set, if q is present it returns the associated page of results back to the user. If q is not contained within the Static Set, then it is looked for in the Dynamic Set. If q is not present, then SDC asks the WSE for the page of results and replaces an entry of the Dynamic Set according to the replacement policy adopted.

The rationale of adopting a static policy, where the entries to include in the cache are statically decided, relies on the observation that the most popular queries submitted to WSEs do not change very frequently. On the other hand, several queries are popular only within relatively short time intervals, or may become suddenly popular due to, for example, un-forecasted events (e.g. the 11^{th} September 2001 attack). The advantages deriving from this novel caching strategy are two-fold. In fact the results of the most popular of the queries can always be usually retrieved from the Static Set even if some of these queries might be not requested for relatively long time intervals. On the other hand, the Dynamic Set of the cache can adequately cover sudden interests of users.

First Level – Static Set. The implementation of the first level of our caching system is very simple. It basically consists of a lookup data structure that allows to efficiently access a set of $f_{static} \cdot N$ entries, where N is the total number of entries of the whole cache, and f_{static} the factor of locked entries over the total. f_{static} is a parameter of our cache implementation whose admissible values ranges between 0 (a fully dynamic cache) and 1 (a fully static cache). The static cache has to be initialized off-line, i.e., with the results of most frequent queries computed on the basis of a previously collected query log.

Each time a query is received, SDC first tries to retrieve the corresponding results from the Static Set. On a cache hit, the requested page of results is promptly returned. On a cache miss, we also look for the query results in the Dynamic Set.

Second Level – Dynamic Set. The Dynamic Set relies on a replacement policy for choosing which pages of query results should be evicted from the cache as a consequence of a cache miss and the cache is full. Literature on caching proposes several replacement policies which, in order to maximize the hit-ratio, try to take the largest advantage from information about recency and frequency of references. SDC surely simplifies the choice of the replacement policy to adopt. The presence of a static read-only cache, which permanently stores the most frequently referred pages, makes in fact recency the most important parameter to consider. Currently, our caching system supports the following replacement policies: *LRU*, *LRU/2* [6] which applies a *LRU* policy to the penultimate reference, *FBR* [8], *SLRU* [5], *2Q* [3], and *PDC* [4] which consider both the recency and frequency of the accesses to cache blocks.

4 Experiments

All the experiments were conducted on a Linux PC equipped with a 2GHz Pentium Xeon processor and 1GB of RAM. Since SDC requires the blocks of the static section of the cache to be preventively filled, we partitioned each query log into two parts: a *training set* which contains 2/3 of the queries of the log, and a *test set* containing the remaining queries used in the experiments. The N most frequent queries of the training set were then used to fill the cache blocks: the first $f_{static} \cdot N$ most frequent queries (and corresponding results) were used to fill the static portion of the cache, while the following $(1 - f_{static}) \cdot N$ queries to fill the dynamic one. Note that, according to the scheme above, before starting the tests not only the static blocks but also the dynamic ones are filled, and this holds even when a pure dynamic cache ($f_{static} = 0$) is adopted. In this way we always starts from the same initial state to test and compare the different configurations of SDC, obtained by varying the factor f_{static}. (i.e. warm cache, using the terminology in [4]).

Figures 2 reports the cache hit-ratios obtained on the *Tiscali* (a), and *Altavista* (b) query logs by varying the ratio (f_{static}) between the sizes of the static and dynamic sets. Each curve corresponds to a different replacement policy used for the dynamic portion of the cache. In particular, f_{static} was varied between 0 (a fully dynamic cache) and 1 (a fully static cache), while the replacement policies exploited were *LRU*, *FBR* [8], *SLRU* [5], *2Q* [3], and *PDC* [4]. The total size of the cache was fixed to 256,000 blocks. Several considerations can be done looking at these plots. First, we can note that the hit-ratios achieved are in some cases impressive, although the curves corresponding to different query logs have different peak values and shapes, thus indicating different amounts and kinds of locality in the query logs analyzed. At a first glance, these differences surprised us. After a deeper analysis we realized that similar differences can also be found by comparing other query logs already studied in the literature [10, 2, 11], thus indicating that users' behaviors may vary remarkably from time to time. Another important consideration is that in all the tests performed SDC remarkably outperformed the other policies, whose performance are exactly those corresponding

to a value of $f_{static} = 0$. The best choice of the value for f_{static} depends from the query log considered and, more importantly, the different replacement policies do not impact heavily on the overall hit-ratio for the optimal values off_{static}.

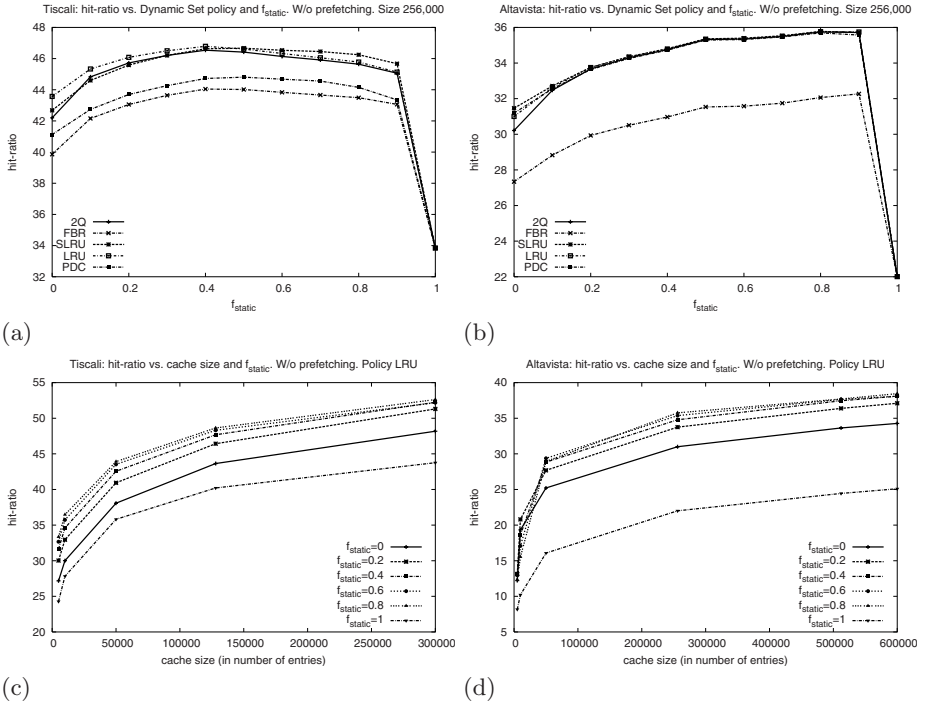

Fig. 2. Hit-ratios achieved on *Tiscali* and *Altavista* logs for different replacement policies and varying f_{static}.

To measure the sensitivity of SDC with respect to the size of the cache, Figures 2 plot the hit-ratios achieved on the *Tiscali* (c) and *Altavista* (d) query logs as a function of the number of blocks of the cache and the f_{static} parameter. As expected, when the size of the cache is increased, hit-ratios increase correspondingly. In the case of the *Tiscali* log, the hit-ratio achieved is about 37% with a cache of $10,000$ blocks, and about 45% when the size of the cache is $50,000$. Note that actual memory requirements are however limited: a cache storing the results as an *Html* page and composed of $50,000$ blocks requires about 200MB of RAM.

We designed our caching system to allow efficient concurrent accesses to its blocks. This is motivated by the fact that a WSE has to process several user queries concurrently. This is usually achieved by making each query processed by a distinct thread. The methods exported by our caching system are thus thread-safe and also ensure the mutual exclusion. In this regard, the advantage of SDC

over a pure dynamic cache is related to the presence of the *StaticTable*, which is a read-only data structure. Multiple threads can thus concurrently lookup the *StaticTable* to search for the results of the submitted query. In case of a hit, the threads can also retrieve the associated page of results without synchronization. For this reason our caching system may sustain linear speed-up even in configurations containing a very large number of threads. Conversely, the *DynamicTable* must be accessed in the critical section controlled by a mutex. Note, in fact, that the *DynamicTable* is a read-write data structure: while a cache miss obviously causes both the associative memory and relative list of pointers to be modified, also a cache hit entails the list pointers to be modified in order to sort the cache entries according to the replacement policy adopted.

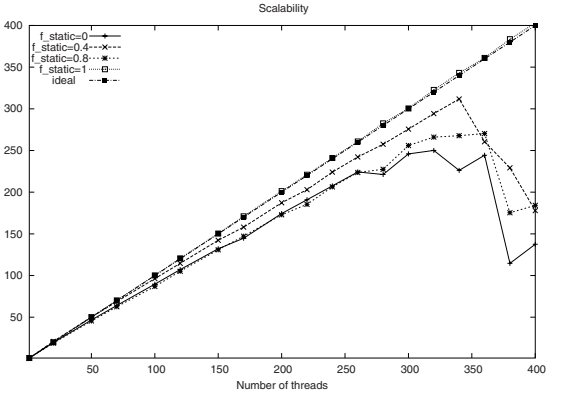

Fig. 3. Scalability of our caching system for different values of f_{static} as a function of the number of concurrent threads used.

Figure 3 shows the performance of our cache system in a multi-threading environment. In particular, the Figure plots, for different values of f_{static}, the scalability of the system as a function of the number of concurrent threads sharing a single cache. Scalability has been measured by considering the ratio between the wall-clock times spent by one and n threads to serve the same large bunch of queries. The replacement policy adopted in running the test was *LRU*. In the case of a cache hit, the thread serving the query returns immediately the requested page of results, and gets another query. Conversely, when a query causes a cache miss, the thread sleeps for 40 ms to simulate the latency of the WSE core in resolving the query. As it can be seen, the system scales very well even when a large number of concurrent threads is exploited. The scalability of a purely static cache is optimal since the cache is accessed read-only, but high scalability values are achieved also when SDC is adopted. Note that even when a purely dynamic cache is adopted ($f_{static} = 0$), our system scales linearly with up to 250 concurrent threads due to the very low cache management times (experimentally measured between 11 and 36μs).

5 Conclusions and Future Works

In this paper we presented SDC, a new policy for caching the query results of a WSE which exploits the knowledge about the queries submitted in the past to make more effective the management of the cache. In particular, we maintain the most popular queries and associated results in a read-only static section of our cache. Only the queries that cannot be satisfied by the static cache section compete for the use of a dynamic cache. The benefits of adopting SDC were experimentally shown on the basis of tests conducted with two large query logs. In all the cases our strategy remarkably outperformed either purely static or dynamic caching policies. We evaluated the hit-ratio achieved by varying the percentage of static blocks over the total, the size of the cache, as well as the replacement policy adopted for the dynamic section of our cache. Moreover, we evaluated cost and scalability of our cache implementation when executed in a multi-threaded environment. The SDC implementation resulted very efficient due to an accurate software design that allowed to make cache hit and miss times negligible, and to the presence of the read-only static cache that reduces the synchronization between multiple threads concurrently accessing the cache.

References

1. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
2. Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Inf. Proc. and Manag.*, 36(2):207–227, 2000.
3. Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. 1994 VLDB*, pages 439–450, 1994.
4. Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proc. of the twelfth international conference on World Wide Web*, pages 19–28. ACM Press, 2003.
5. Evangelos P. Markatos. On caching search engine results. In *Proc. of the 5th Int. Web Caching and Content Delivery Workshop*, 2000.
6. Elisabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffer. In *Proc. of the 1993 ACM SIGMOD International Conference On Management Of Data*, pages 297–306, 1993.
7. S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. In *In proc. of ParCo 2001 int'l conf.*, 2001.
8. 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.
9. P.C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engine. In *SIGIR'01*, 2001.
10. C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. In *ACM SIGIR Forum*, pages 6–12, 1999.
11. A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. Searching the web: the public and their queries. *J. Am. Soc. Inf. Sc. & Tech.*, 53(2):226–234, 2001.