A Hybrid Strategy for Caching Web Search Engine Results

T. Fagni[1], S. Orlando[2], P. Palmerini[12], P. Perego[1], F. Silvestri[1]

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

March 26, 2003

Abstract

This paper discusses the design and implementation of an efficient caching system aimed to exploit the locality present in the queries submitted to a Web search engine. Previous works showed that there is a significative temporal locality in the queries, and demonstrated that caching query results is a viable strategy to increase search engine throughput. We enhance previous proposals in several directions. First we propose the adoption of a hybrid strategy for caching, where the results of the most frequently submitted queries are maintained in a static cache of fixed size, and only the queries that cannot be satisfied by the static cache compete for the use of a dynamic cache. We experimentally demonstrate the superiority of our hybrid strategy over a purely static or dynamic caching policy by evaluating the hit-rate achieved on three large query logs by varying the size of the cache, the percentage of static cache entries, and the replacement policy used for managing dynamic cache entries. Moreover, we show that search engine query logs also exhibit spatial locality, since users often require subsequent pages of results for the same query. Our caching system also take advantage of this type of locality by exploiting a sort of adaptive prefetching strategy. Finally, differently from other works, we accurately evaluate cost and scalability of our cache implementation.

Categories and Subject descriptors: C.4 [Performance Of Systems]: Modeling techniques; D.1.3 [Concurrent Programming]: Parallel programming; H.3.3 [Information Search and Retrieval]: Search Process; H.3.4 [Systems and Software]: Performance evaluation (efficiency and effectiveness); H.3.5 [Online Information Services]: Web-based services; I.6 [Simulation and Modeling]: Model Validation and Analysis.

Keywords: Caching, search engines, query log analysis, performance

1 Introduction

Caching is a very effective technique to make scalable a service that distributes data/information to a multitude of clients. As suggested by many researchers, caching can also be used to improve the efficiency of a Web Search Engine (WSE) [13, 4, 8]. This is motivated by the high locality present in the stream of queries processed by a WSE, and by the relatively infrequent updates of WSE indexes that allow us to think of them as mostly read-only data.

Nowadays, WSEs are commonly used to find information and navigate in the Web. In the last years we have observed an impressive growth in the number of pages indexed as well as in the number of queries submitted to the most popular WSEs, thus requiring WSEs to be designed in a scalable way. To improve the scalability we could replicate or decentralize WSE. For example, several researchers think that the growth rate of the Web should suggest that the centralized approach followed till now in the design of WSEs, i.e. centralized crawling/indexing/querying activities, is no longer adequate, while a decentralized P2P paradigm, according to which a multitude of independent small WSEs might collaborate, should favor scalability. However, even if a P2P approach is adopted, to cache query results should still be a valuable technique to improve performance and scalability.

WSE 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 advantage of improving the shareness of

query results among different users. Moreover, this kind of caching has the effect of reducing I/O, network and computation bandwidths exploited by the query core services of a WSE to prepare the page of relevant results to be returned to a user. For example, consider that, in order to prepare a page, we have to intersect inverted lists that can be distributed, and to globally rank the results to decide which are the most relevant ones. One of the issues related to a server-side cache is the limited resources usually available on the WSE server, in particular the RAM memory used to store the cache entries. However, the architecture of a modern WSE may be very complex and includes several machine, since a query may require various sub-tasks to be executed on these machines [6]. For example a large and scalable WSE, which is typically placed behind an http server, is usually composed of several searcher modules, each of which preferably runs on a distinct machine and is responsible for searching the index relative to one specific sub-collection of documents. Of course, in front of these searcher machines there must exist a mediator/broker, which collects and reorders the results of the various searchers, and produces a ranked vector of the most relevant document identifiers (DocIDs), e.g. a vector composed by 10 DocIDs. These DocIDs are then used to get the associated URLs and page snippets to include in the html page returned to the user through the http server. Within this architecture the RAM memory is a very precious resource for the machines that host the searchers, which perform well only if the mostly accessed sections of their huge indexes can be stored 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.

Depending on the memory available, we could devise a cache whose blocks stores complete html pages (*Html cache*), or a cache that simply stores the DocID vectors (*DocID cache*). Note that, given a fixed memory size, the *DocID cache* could include a huger amount of blocks than an *Html cache*, since in that case each block is a small vector of integer identifiers. Conversely, a hit in a *DocID cache* only returns a vector of DocIDs, while the html page to be delivered to the user has still to be prepared.

In this paper, we are interested in studying effective policies and implementations for server-side WSE caches. In particular, we will analyze the behavior of a one-level cache, either *Html* or *DocID cache*, in terms of miss and hit rate. Note that if the WSE logs were collected on a proxy, we could use the same analysis and implementation discussed in this paper to deploy a proxy-side *Html cache*.

We propose a novel hybrid replacement policy to adopt in the design of our cache, and also discuss an actual implementation of the cache, which can be concurrently accessed by several threads managing distinct user queries, and whose scalability has been accurately assessed. According to our hybrid caching strategy, the results of the most frequently accessed queries are maintained in a static cache of fixed size, which is completely rebuilt at fixed time intervals. Only the queries that cannot be satisfied by the static cache compete for the use of a dynamic cache. Note that several complex cache replacement policies have been previously proposed. These policies try to find a trade-off between two criteria: the recency of references and their global frequency. Our hybrid cache represents an effective and fast way to address both criteria. While the static cache maintains results of queries that are globally frequent, a simple and fast policy like LRU, which only takes into account query reference recency, could be adopted for the dynamic cache.

We experimentally demonstrated the superiority of our hybrid strategy over a purely static or dynamic caching policy, by evaluating the hit-rate achieved on three large query logs by varying the size of the cache, the percentage of static cache entries, and the replacement policy used for managing dynamic cache entries. Moreover, we showed that WSE query logs exhibit not only temporal locality, but also a limited spatial locality, due to requests for subsequent pages of results. Our caching system also addresses spatial locality by exploiting a sort of adaptive prefetching strategy. While simple server-side caching surely reduces the load over the core query service of a WSE and improves its throughput, prefetching aims to increase the cache hit rate and thus the responsiveness of the WSE, but may involve a larger load over the same core query service. So an accurate study of trade-offs of prefetching is necessary, and we addressed this in the experimental section of the paper by analyzing pros and cons of various prefetching strategies.

The rest of this paper is organized as follows. Section 2 describes the query logs used and discusses the results of the preliminary statistical analysis conducted on their contents. In Section 3 we describe in depth our hybrid approach to Web search engine result caching. Section 4 presents and discusses the results of the experiments conducted, and, finally in Section 5 we draw future work and some conclusions.

Table 1: Main characteristics of the query logs used.

Query log	queries	distinct queries	date
Tiscali1	1,352,079	626,885	March 2001
Tiscali2	3,278,211	1,538,934	April 2002
Excite	2,475,684	1,598,908	September 1997

2 Analysis of the query logs

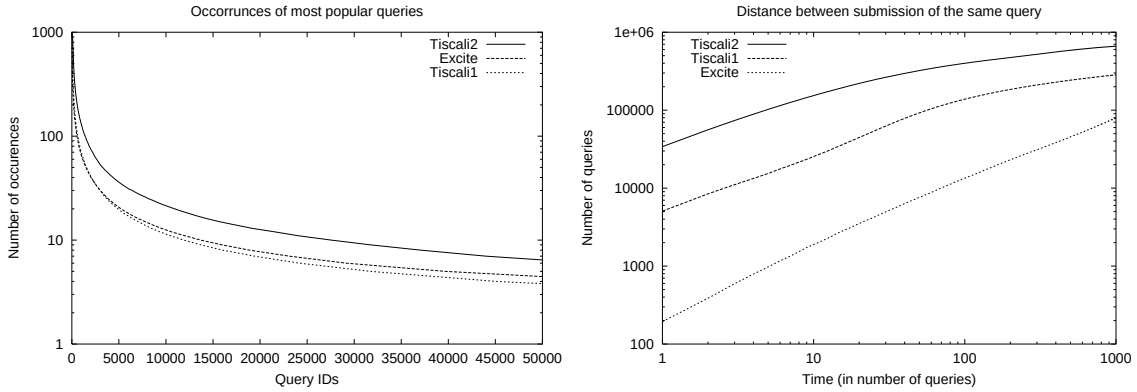
In order to evaluate the behavior of different caching strategies we used query logs from the Tiscali and EXCITE search engines. In particular we used *Tiscali1* and *Tiscali2*, two different traces of the queries submitted to the Tiscali WSE engine (www.janas.it) on March 2001 and April 2002, respectively, and *Excite*, a publicly available trace of the queries submitted to the EXCITE WSE (www.excite.com) on September 26^{th} 1997. Table 1 reports the main characteristics of the query logs used. While about the 46% of the total number of queries appearing in the relatively recent Tiscali logs are distinct, in the Excite log this percentage increases up to 64%. Therefore, only looking at the numbers of distinct queries appearing in the three logs, we could deduce that the locality found in the Excite log, i.e. the oldest one, might be less than in the Tiscali ones, since only the 36% (54% in the Tiscali logs) of all its queries corresponds to re-submissions of previously submitted queries.

Each entry of a query log refers to a single word-based 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 case-folding and by removing stop words and fields not useful for our purposes. 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. A page is always composed of 10 results. In particular, since a WSE globally reorders the results according to a given rank, the top 10 results will be included in page 1, the next 10 results in page 2, and so on. Indeed, the Tiscali logs contains queries in which, in principle, it is possible to select arbitrary intervals within the globally ordered sequence of results. In practice, however, except for a few cases that were normalized, the queries has the common page-based form illustrated above.

2.1 Temporal locality

The plots reported in Figure 1 assess the temporal locality present in the query logs. 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 three 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 *Tiscali1* and *Tiscali2* logs for more than $100,000$ times, the time interval between successive submissions of the same query is less than 100; in the case of the *Excite* log we encountered a lower temporal locality. However, also in this log for more than $10,000$ times a given query is repeated at a distance lower than 100.

We think that the lower locality present in the *Excite* log is mainly due to its oldest age. It contains several long queries expressed in natural language like *"Where can I find information on Michael Jordan"*. In the query above the first six terms are meaningless, while the only informative keywords are the last two. Such kind of queries can be considered a symptom of the poor capacity of the users to interact with a WSE five years ago. Although we tried to clean the *Excite* queries by removing *stopwords*, i.e. by eliminating meaningless terms like articles, adverbs and conjunctions, the query above still resulted syntactically different from the simpler query *"Michael Jordan"* even if the same results should be considered relevant from both the queries.



(a) (b)

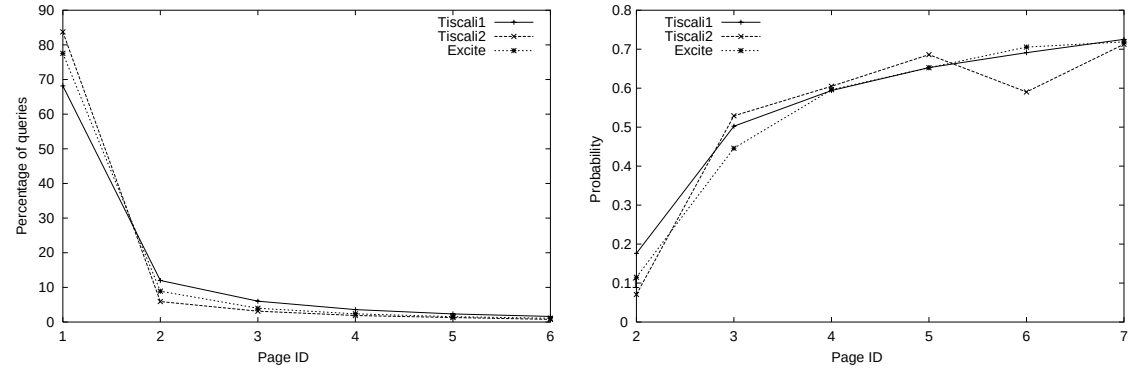
Figure 1: (a) Number of submissions of the most popular queries contained in the three query logs. (b) Distances (in number of queries) between subsequent submissions of the same query.

2.2 Spatial locality

Several works analyze WSE query logs in order to determine behaviors and preferences of users [1, 9, 11, 2, 10]. Although different in some assumptions and in several conclusions, these works highlight that WSE users in most cases submit short queries and visit only a few pages of results. Estimations reported in these works differ, in some cases, remarkably: depending on the query log analyzed, percentages ranging from 28% to 85% of user searches only require the first page of results, so that we can expect that from 15% up to 72% of user searches retrieve two or more pages of results. This behavior involves the presence of significant spatial locality in the stream of queries processed by a WSE. Given a query requesting a page of relevant results with respect to a list of keywords, with high probability the WSE will receive a request for one of the following pages within a small time interval. To validate this consideration we measured the spatial locality present in our query logs. Figure 2 reports the results of this analysis. In particular, Figure 2.(a) plots the percentage of queries in each log file as a function of the index of the requested page. As expected, most queries require the first page of results. On the other end, while there is a huge difference between the number of queries requesting the first and the second page, this difference becomes less significative when we consider the second and the third page, the third and the fourth, and so on. This may reflect different usages of the WSE. When one submits a focused search, the relevant result is usually found in the first page. Otherwise, when a generic query is submitted, it is necessary to browse several pages in order to find the relevant information. To highlight this behavior, Figure 2.(b) plots the probability of the occurrence of a request for the i-th page, given a previous request for the $(i-1)$-th one. As it can be seen, this probability is low for $i = 2$, while it increases remarkably for higher values of i.

3 Our hybrid cache

Markatos already studied pure static caching policies for WSE results, and compared them with dynamic caching ones [4]. The rationale of adopting a static policy, where the entries to include in the cache are statically decided, relies on the observation 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). These considerations suggested us to adopt a hybrid (static+dynamic) strategy for caching query results, where the results of most popular queries are maintained in a static cache, and only the queries that cannot be satisfied with the content of the static cache compete for dynamic cache entries. The advantages deriving from this novel caching strategy are twofold. Firstly, hybrid caching may enhance performances. In fact the management of the static part of the cache is costless, and its entries can be accessed concurrently without synchronization in a multi-threading environment. Secondly, a hybrid strategy may achieve the main benefits of both static and dynamic caching. In fact:



(a) (b)

Figure 2: Spatial locality in the three query logs analyzed: (a) percentage of queries as a function of the index of the page requested; (b) probability of the occurrence of a request for the i-th page given a previous request for page $(i-1)$.

- the results of the most popular of the queries can always be retrieved from the static cache even if some of these queries might be not requested for relatively long time intervals;
- the dynamic part of the cache can adequately cover sudden interests of users.

In the following we discuss the organization of our caching system and we briefly describe the caching policies implemented whose evaluation is the subject of Section 4.

3.1 Static cache

The static part 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 total number of entries of the whole cache, and f_{static} the factor of static 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). Static cache can be initialized off-line on the basis of a previously collected query log, and usually contain the most frequent queries found in the log.

Each time a query is received, our caching system first tries to retrieve the corresponding results from the static cache section. On a cache hit, the requested page of results is promptly sent back to the user. On a cache miss, we also look for the query results in the dynamic cache section.

3.2 Dynamic cache

Dynamic caching involves the adoption of a replacement policy for choosing which pages of query results should be evicted from the cache whenever, as a consequence of a cache miss, new pages returned by the search engine has to be cached and the cache is full. Literature on caching proposes several replacement policies which, in order to maximize the hit-rate, try to take the largest advantage from information about recency and frequency of references. Our hybrid caching strategy surely simplifies the choice of the replacement policy to adopt. The presence of a static cache, which permanently stores most frequently referred pages, makes in fact recency the most important parameter to consider. As a consequence, some sophisticated (and often computationally expensive) policies specifically designed to exploit at the best both frequency and recency of references are probably not useful in our case. However, since we want to demonstrate the advantage of our hybrid policy over pure dynamic or static ones, we also implemented some of these sophisticated replacement policies.

Currently, our caching system supports the following replacement policies: *LRU*, *LRU/2* [5] which applies a LRU policy to the penultimate reference, *FBR* [7], *LRU-2S* [4], and *2Q* [3], which consider both the recency and frequency of the accesses to a page.

The choice of the replacement policy to be used is performed at start-up time, and clearly affects only the management of the $(1 - f_{static}) \cdot N$ dynamic entries of our caching system.

3.3 Prefetching query results

The spatial locality present in the stream of queries submitted to the WSE can be exploited by anticipating the request for the following pages of results. In other words, when our caching system processes a query of the form *(keywords, page_no)*, and the corresponding page of results is not found in the cache, it might forward to the core query service WSE an expanded query, requesting k consecutive pages starting from page *page_no*, where $k \geq 1$ is the *prefetching factor*. To this end, the queries that arrive at the WSE cache system are thus expanded, and are passed to the core query service of the WSE as *(keywords, page_no, k)*. Note that, according to this notation, $k = 1$ means that prefetching is not activated.

When the results of the expanded queries are returned to the caching system, the retrieved pages are stored into k distinct blocks of the cache, while only the first page is returned to the requesting user. In this way, if a query for a following page is received within a small time interval, its results can surely be found into the cache. Prefetching clearly involves additional load on the WSE, although the cost of resolving a query is logarithmic with respect to the number of results retrieved [12]. Moreover, an aggressive prefetching strategy might negatively affect the replacement policy adopted, since a miss causes the replacement of several pages with new pages till not accessed.

In order to maximize the benefits of prefetching, and, at the same time, reduce the additional load on the WSE, we can take advantage from the characterization of the spatial locality present in the logs discussed in Section 2. Figure 2.(b) shows that, given a request for the i-th page of results, the probability of having a request for page $(i + 1)$ in the future is about 0.1 for $i = 1$, but becomes approximately 0.5 or greater for $i > 1$. Therefore, an effective heuristic to adopt is to prefetch additional pages only when the cache miss has been caused by a request for a page different from the first one. In this way, since the prefetched pages will be actually accessed with sufficiently high probability, we avoid to fill the cache with pages that are accessed only rarely and, at the same time, we reduce the additional load on the core query service of the WSE. Our caching system thus adopts this simple heuristic, and adaptively changes the prefetching factor k as a function of the index of the page requested. In particular, if K_{MAX} is the maximal prefetching factor, and *(keywords, page_no)* is a generic query that caused a *cache-miss*, the prefetching factor k for the expanded query *(keywords, page_no, k)* is chosen as follows:

$$k = \begin{cases} K_{MAX} & \text{if } page\_no > 1 \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

In Section 4 we will evaluate the efficacy of this prefetching technique, and discuss the tuning of the prefetching factor.

3.4 Architecture of our Caching System

The implementation of complex software systems should consider a number of aspects that rarely can coexists together. These aspects can be collected into two broad categories: *Software Quality*, and *System Efficiency*. Typically, the criteria used to evaluate the "*quality*" are: *modularity*, *flexibility*, and *maintainability*. While, when considering "*performance*", one is more interested in optimizing the *response time*, the *scalability*, and the *system throughput*.

In designing our hybrid caching system we tried to combine the two aspects of Software Quality and System Efficiency into a C++ implementation, which heavily relies on the use of different Object Oriented (OO) techniques. For example, to improve system portability we used two multi-platform OO toolkits *STLPort* and *ACE*, which are highly optimized and thus ensure high performances. Moreover, in terms of system flexibility, we implemented our software by heavily using C++ features such as inheritance and templates. In particular, the use of templates permitted us to abstract from the specific data types being stored, and allows our system to be easily adapted to different formats. For instance, one can choose to store complete html pages of results, which usually contain URLs, snippets, titles, etc. along with formatting information (*Html cache*); otherwise, one can store only the numbers used in the WSE to identify the relevant documents corresponding to a given query ((*DocID cache*).

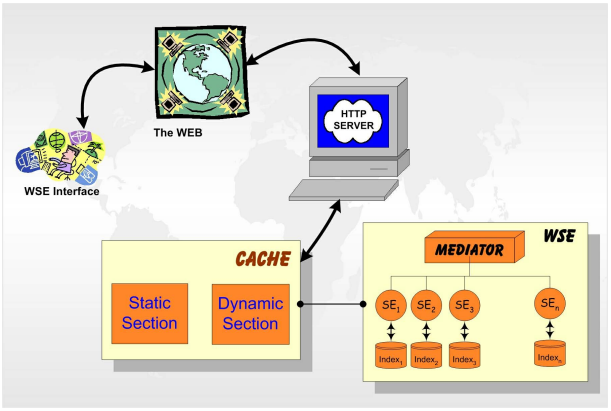


Figure 3: Typical cache placement within the query service of a WSE.

As we anticipated in the introduction, our cache can easily be integrated into a typical WSE operating environment. A possible logical placement of our cache, as shown in Figure 3), is between the Web server and the mediator. When the mediator is responsible also for the preparation of the complete html page, the cache shown in the Figure should be an *Html cache*. Otherwise, the cache can be a *DocID cache* too.

In the following we will describe the modules composing our system. In particular, we will focus on the description of the data structures we used to efficiently store and retrieve the elements contained into the cache.

Cache Data Structures. Our cache is fully associative, i.e. a query result may be associated with any entry of the cache. It is accessed through user queries, whose normalized format is *(keywords, page_no)*. To ensure efficiency in the query lookup, we used a hash function which transforms the key (i.e., the user query) into the right table address. In case of cache hits we can thus perform this lookup-and-retrieve task in $O(1)$ time complexity.

The static section of our hybrid cache, hereinafter *StaticTable*, is stored as a simple associative memory that maps queries into their associated results. The *DynamicTable*, which is the dynamic section of the cache, not only implements the associative memory, but also maintains one o several orderings among the various cache entries. The specific orderings depend on the replacement policy implemented. When a cache miss occurs and no empty cache entries are available, these orderings are exploited to select the cache entry to be replaced. For example, if we adopt an LRU replacement policy, we have to select the least recently used cache entry, and thus the only ordering to maintain regards the recency of references. In practice, the *DynamicTable* implements these orderings through pointers used to interlink the cache entries, thus maintaining sorted lists. This avoids to copy the data entries of *CacheLookup* table when an ordering must be modified.

Concurrent Cache System. We designed our caching system to permit the concurrent access by multiple threads. This is motivated by the fact that several user queries may be processed concurrently by a WSE, so that we assume that each of them is processed by a distinct thread. The methods exported by our caching system are thus thread-safe and also ensure the mutual exclusion. The advantages of our concurrent hybrid cache are related to the presence of the *StaticTable*, which is a read-only data structure, while 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 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. Obviously, whenever a thread finds the submitted query into the StaticTable, it needs not to compete in accessing the dynamic (read-write) portion of the cache. For this reason our caching system may sustain linear speed-up even in configurations containing a very large number of threads (> 100).

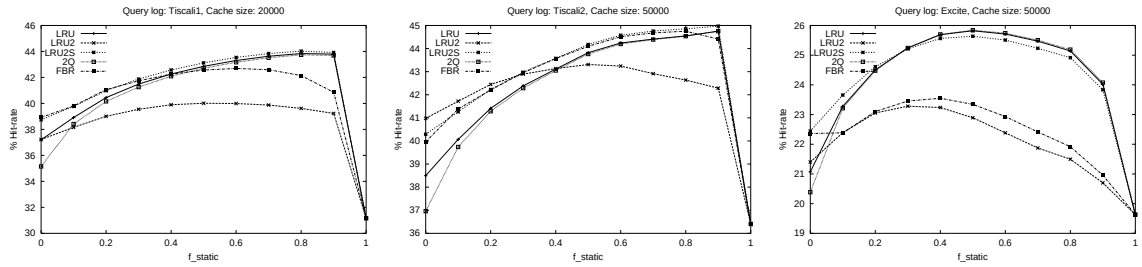
4 Experimental results

In order to experimentally evaluate efficiency and efficacy of the proposed hybrid caching system, we had to simulate its inclusion between a WSE http server and a WSE core query service (see Figure 3). In the tests conducted, the behavior of a multi-threaded WSE http server, which forwards user queries to the cache system and waits for query results, was simulated by actually reading the queries from a single log file. The WSE core query service, which is invoked to resolve those queries that have caused cache misses, was simply simulated by sleeping the process (or thread) which manages the query for a fixed amount of time.

In the following, the cache size will always refer to its total number of blocks N. However, as discussed in Section 1, our system can implement both an *Html cache* and a *DocID cache*. So, the actual memory occupation of a block belonging an *Html cache* can be estimated as $4KB$, where $4KB$ is the typical size of a dynamic html page returned by a WSE and including 10 results. Conversely, the actual memory occupation of a block belonging to a *DocID cache*, and containing the same number of results, is approximately $40 \div 80$ Bytes.

Since our hybrid caching strategy 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 $f_{static} \cdot \cdot N$ most frequent queries (and corresponding results) were used to fill the static portion of the cache, while the following $(1 - f_{static}) \cdot \cdot N$ 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 various possible configuration of our hybrid cache, obtained by varying the factor f_{static}. Finally, all the experiments were conducted on a Linux PC equipped with a 2GHz Pentium Xeon processor and 1GB of RAM.

Hybrid caching. Figure 4 reports cache hit rates obtained on the query logs *Tiscali1*, *Tiscali2*, and *Excite* by varying the ratio (f_{static}) between static and dynamic cache entries. 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*, *LRU/2*, *FBR*, *LRU-2S*, and *2Q*. In these tests prefetching was not exploited, and the total size of the cache was fixed to $20,000$ blocks for the smallest *Tiscali1* query log, and to $50,000$ blocks for the other two query logs.



(a) (b) (c)

Figure 4: Hit rate obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries: (a) *Tiscali1*, (b) *Tiscali2*, (c) *Excite*.

Several considerations can be done looking at the three plots. First, we can note that the hit rates achieved are in some cases impressive, although the curves corresponding to different query logs have different shapes, thus indicating different amounts and kinds of locality in the query logs analyzed. At 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 [1, 9, 11, 2, 10], thus indicating that users' behaviors may vary remarkably.

Another important consideration is that in all the tests performed our hybrid caching strategy remarkably outperformed either purely dynamic or static policies. The best choice of the value for f_{static} depends from the query log considered and, in some cases, from the replacement policy adopted. In general we can say that for the *Tiscali1* and *Tiscali2* logs, which exhibit higher locality than the *Excite* one (see Section 2), a

cache organization where the static portion predominates allows highest hit rates to be obtained. For the *Excite* query log the best choice is instead to have about an half of the cache blocks managed statically and an half dynamically.

For what regards the replacement policies, the tests demonstrated that while for cache organizations where the number of dynamic cache blocks predominates (i.e., small values of f_{static}), the *LRU-2S*, *FBR*, or *LRU/2* replacement policies outperform *LRU* and *2Q*, the opposite often holds when the percentage of static cache blocks is increased. This behavior is motivated by the presence of the static portion of our cache, which already stores the results of the most frequently referred queries in the past. This seems to penalize replacement policies that consider both recency and frequency of the accesses to a page. As a consequence, a simple (and computational inexpensive) replacement policy like *LRU*, which only considers recency, can be profitably adopted within our hybrid cache. Figure 5 plots the hit rates achieved on the *Tiscali2* and *Excite* query logs as a function of the number of blocks of the hybrid cache. To perform the test on the *Tiscali2* and *Excite* logs, we set f_{static} to 0.8. As expected, when the size of the cache is increased, hit rates increase correspondingly. In the case of the *Tiscali2* log, the hit rate 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: an *Html cache* of $50,000$ blocks requires about 200MB of RAM, while a *DocID cache* of the same size requires less than 5MB!

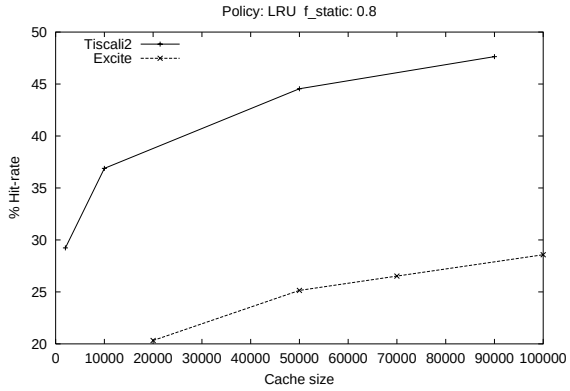


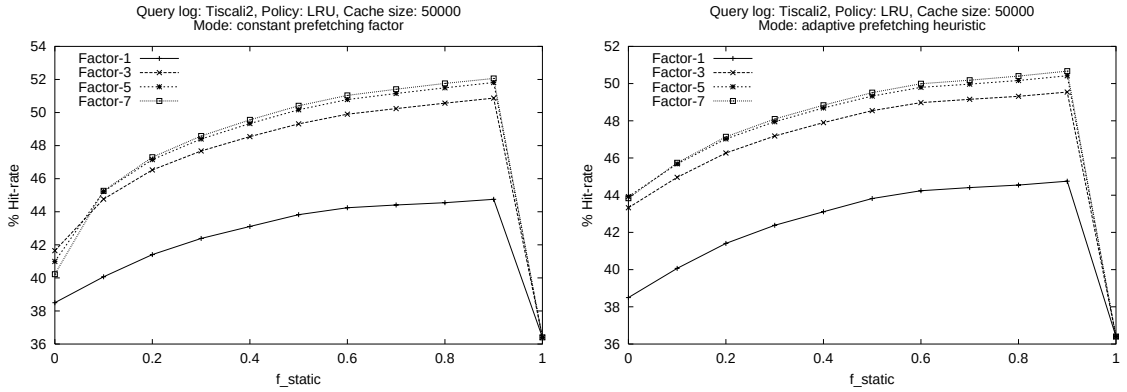
Figure 5: Hit rates achieved on the *Tiscali2* and *Excite* logs as a function of the size of the cache.

Prefetching. Figure 6 shows the hit rates achieved on the *Tiscali2* query log by varying not only the f_{static} ratio, but also the prefetching factor k, whose tested values were 1 (no prefetching), 3, 5 and 7. The replacement policy adopted for the dynamic portion of the cache was always LRU.

Figure 6.(a) refers to tests conducted by maintaining the prefetching factor k *constant* for all the pages requested. Conversely, the curves reported in the plot of Figure 6.(b) are relative to tests conducted by prefetching additional pages only when a cache miss is caused by a request regarding a page of results different from the first one, i.e., by employing the *adaptive* heuristic discussed in Section 3.

From both the plots we can see that prefetching is able to effectively exploit spatial locality present in the logs: we obtained remarkable improvements in the cache hit rate. On the other hand, by comparing the curves plotted in Figure 6.(a) and 6.(b), we can see that while the constant prefetching slightly outperforms the adaptive prefetching heuristic when our hybrid strategy is used, the opposite holds for a pure dynamic cache. However, this higher hit rate achieved by exploiting constant prefetching must be paid with an huge increase in the load on the core WSE query service. By profiling execution, we measured that for $f_{static} = 0.8$, in the case of a constant prefetching factor 3, only 5.7% of the prefetched pages are actually referred by the following queries, while when the adaptive prefetching heuristic is adopted (with $K_{max} = 3$), this percentage increases up to 46%. We can thus conclude that the proposed adaptive heuristic constitutes a good trade-off between the maximization of the benefits of prefetching, and the reduction of the additional load on the WSE.

Multi-threading. Figure 7 shows the performance of our cache system in a multi-threading environment. In particular, the Figure plots the scalability of the system as a function of the number of concurrent threads



(a) (b)

Figure 6: Hit rate for different prefetching factors as a function of the ratio between static and dynamic cache entries: (a) with constant prefetching factor, and, (b) with adaptive prefetching factor.

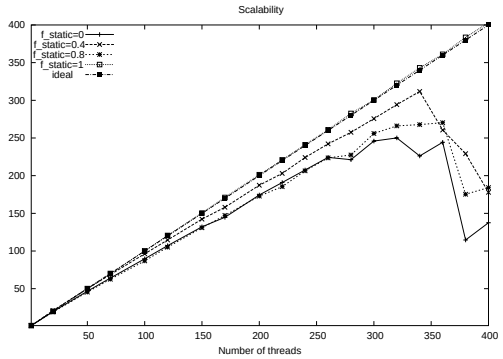


Figure 7: Scalability of our caching system as a function of the number of concurrent threads used.

sharing a single cache. Scalability has been measured by considering the wall-clock times spent by one and n threads to serve the same large bunch of queries. As it can be seen, the system scales very well even with a large number of concurrent threads. The scalability of a purely static cache is obviously near-optimal since the cache is accessed read-only, but high scalabilities are achieved also when hybrid organizations are adopted. Note that even when a purely dynamic cache is adopted, our system scales linearly with up to 180 concurrent threads. This is mainly due to the accurate software design: with a cache of $50,000$ blocks, $f_{static} = 0.8$, no prefetching, average cache hit and miss management times are only $11\mu s$ and $36\mu s$, respectively (neglecting the WSE latency for misses).

5 Conclusions

In this paper we presented a new hybrid policy for caching the query results of a WSE. The main enhancement over previous proposals regards the exploitation of past knowledge about queries submitted to the WSE to make more effective the management of our cache. In particular, since we noted that the most popular queries that are submitted to a WSE do not change very frequently, we maintain these queries and associated results in a read-only static section of our cache. The static section is updated at regular times on the basis of the WSE query log. Only the queries that cannot be satisfied by the static cache section compete for the use of a dynamic

cache.

Our hybrid caching strategy, based on a static+dynamic cache, simplifies the choice of the replacement policy to adopt in the dynamic section of the cache. The presence of a static cache, which permanently stores the most frequently referenced pages, makes in fact *recency* the most important parameter to consider. As a consequence, a simple LRU policy handles effectively block replacement in our dynamic cache section, while some sophisticated (and often computationally expensive) policies specifically designed to exploit at the best both *frequency* and *recency* of references result less effective.

The benefits in adopting our hybrid caching strategy were experimentally shown on the basis of tests conducted with three large query logs. In all the cases our strategy remarkably outperformed purely static or dynamic caching policies. We evaluated the hit-rate 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 showed that WSE query logs also exhibit spatial locality. Users, in fact, often require subsequent pages of results for the same query. Our caching system takes advantage of this locality by exploiting a sort of adaptive prefetching strategy. We accurately evaluated pros and cons of this adaptive strategy, where pros regard the hit rate improvements, while cons are concerned with the higher load over the core query service of the WSE.

Finally, differently from other works, we evaluated cost and scalability of our cache implementation when executed in a multi-threaded environment. Our hybrid cache 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.

A future work regards the evaluation of a two-level cache on the server-side, with an *Html cache* at level L1 (placed on the machine that hosts http server), and a *DocID cache* at level L2 (placed on the machine that hosts the mediator). Moreover, we would like to evaluate the effectiveness of our technique for caching WSE query results at the proxy-side, on the basis of locally collected query logs. Since the users of a proxy generally belong to a homogeneous community (e.g. the departments of a University), we expect to find higher locality in the stream of queries submitted. Moreover, exploiting caching at this level both enhances user-perceived *QoS* and saves network bandwidth.

Acknowledgements

We acknowledge the financial support of the *Fondazione Cassa di Risparmio di Pisa* (funding the WebDigger project) and of the *Italian Ministry of University and Research* (funding the ECD project). Moreover, we want to thank Antonio Gullí for several useful discussions and Ideare S.p.A. for making the Tiscali query logs available to us.

References

[1] C. Hoelscher. How internet experts search for information on the web. Paper presented at the World Conference of the World Wide Web, Internet, and Intranet, Orlando, FL, 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. *Information Processing and Management*, 36(2):207–227, 2000.

[3] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. 1994 Very Large Data Bases*, pages 439–450, 1994.

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

[5] Elisabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffer. In *Proceedings of the 1993 ACM Sigmod International Conference On Management Of Data*, pages 297–306, 1993.

[6] S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. In *The 2001 Parallel Computing Conference (ParCo 2001)*, 2001.

[7] 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.

[8] 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.

[9] 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.

[10] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. Searching the web: the public and their queries. *J. Am. Soc. Information Science and Technology*, 53(2):226–234, 2001.

[11] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. From e-sex to e-commerce: Web search changes. *Computer*, 35(3):107–109, March 2002.

[12] Ian H. Witten, Alistar Moffat, and Timothy C. Bell. *Managing Gigabytes – Compressing and Indexing Documents and Images*. Morgan Kaufmann Pubilshers Inc., 1999.

[13] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of IEEE INFOCOM 2002, The 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002.