A Popularity-driven Caching Scheme for Meta-search Engines: An Empirical Study+

Sang Ho Lee[1], Jin Seon Hong[1], and Larry Kerschberg[2]

[1] School of Computing, Soongsil University, Korea
{shlee, jshong}@computing.soongsil.ac.kr
http://orion.soongsil.ac.kr/

[2] E-Center for E-Business, Department of Information and Software Engineering
George Mason University, U.S.A.
kersch@gmu.edu
http://eceb.gmu.edu/

Abstract. Caching issues in meta-search engines are considered. We propose a popularity-driven cache algorithm that utilizes both popularities and reference counters of queries to determine cache data to be purged. We show how to calculate query popularity. Empirical evaluations and performance comparisons of popularity-driven caching with the least recently used (LRU) and least frequently used (LFU) schemes have been carried out on collections of real data. In almost all cases, the proposed replacement policy outperforms LRU and LFU.

1 Introduction

Web search engines enable Internet users to search the web. Yet, no single search engine indexes all the information available on the Internet, and the coverage of a single search engine is surprisingly low [5]. A number of meta-search engines have been developed to augment the low coverage of a single search engine [9]. A meta-search engine accesses external search engines to get search results when the user issues a query.

Accessing external search engines in response to user queries is likely to lead to long response times for end-users, as network traffic and slow remote servers can lead to long delays in results delivery. One approach to overcome the slow response time is to cache previously retrieved results, called results cache [1], for possible future reference. In this case, meta-search engines maintain results of search engines (i.e., a collection of HTML pages) for possible use to answer future queries.

Caching is a quite well known technique in various fields of Computer Science and has been studied extensively. Among others, page caching, tuple caching and semantic caching schemes [2, 3, 6] are used in practice. An essential part of caching

---

+ This work was supported by grant No. (2000-2-51200-002-3) from the Basic Research Program of the Korea Science and Engineering Foundation, and by the E-Center for E-Business, the Virginia Center for Innovative Technology.

techniques are the cache replacement policies [4, 7, 8], which decide which part of data in the cache will be deleted to accommodate new data. The most popular policy is the least recently used (LRU), which replaces a block that has been in the cache longest with no references to it. The least frequently used (LFU) policy replaces a block that has experienced the fewest references in the cache.

All the above block (page) replacement policies have been studied in the context of operating systems or database systems. They cannot continue to work well in meta-search engines, since circumstances of meta-search engines are quite different. First, query results of search engines do not need to be written back to a search engine's memory. Data consistency issues, which are very important in operating systems and database systems, do not apply in the meta-search context. Second, results of search engines are apt to be stale after a certain period of time has passed since the data was collected. Search engines are not informative or even friendly enough to let users know the staleness of search results. Lastly, results caching is important for meta-search engines, because the underlying search engines may not be accessible to process a query.

A caching scheme in a meta-search engine is considered in this paper. We describe the cache scheme of our meta-search engine, which is being developed as a research vehicle, in terms of its architecture, algorithms, and operational flow. In particular, we propose a popularity-driven cache algorithm that utilizes both popularities and reference counters of queries to determine cached data to be replaced. We show how to calculate query popularity. Empirical evaluations and performance comparisons of popularity-driven caching with commonly used LRU and LFU have been carried out on collections of real data. In almost all cases, the proposed replacement policy outperforms LRU and LFU.

The rest of this paper is organized as follows. Section 2 describes the cache organization of our experimental meta-search engine. The design philosophy is also described. In section 3, we present the notions of popularity, popularity collection and calculation, and a replacement policy based on the popularity of queries. Empirical evaluations are found in section 4, along with our analysis. Section 5 contains closing remarks and future work.

2 Preliminaries

This section briefly describes a meta-search engine, named EzFinder, which is being developed as a research vehicle (Figure 1). EzFinder implements a wrapper-level cache scheme in which each wrapper has its own cache to hold data. The cache in each wrapper stores query results (here, HTML pages) along with queries. Caching in the wrapper could be done in two places: in main memory and in disk. This paper considers caching data in main memory only.

The EzFinder query manager accepts user queries and transforms them into queries in the internal format of each search engine. The transformed queries are transferred to the wrapper manager, which in turn dispatches sub-queries to each wrapper. Each wrapper connects to external search engines, and receives search results from them. All collected query results are transferred to the output manager for post-processing.

The output manager processes (such as collating, ranking, etc.) all returned results and presents them to users.

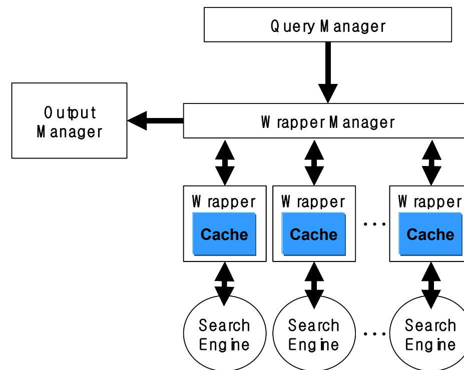

Fig. 1. Cache in EzFinder

We analyzed user queries, which were over 781,550 queries, collected at a popular search engine in Korea in which virtually all queries were issued in Korean. More than 86% of all queries consist of a single word. We have decided that EzFinder shall store query results of only single word queries in the cache. Query results of user queries with more than one word are not stored in the cache, so in these cases wrappers always connect to search engines to get answers.

At this point, we alert readers to language differences between Korean and English. From a syntactic view, words in both languages are delimited by blanks, but the semantic expressiveness of one word is quite different. A single word in Korean often denotes multiple-word expressions in English. For example, "information retrieval" and "computer science" use two words in English, but the corresponding Korean expressions could have only one word. Therefore, the one-keyword query assumption is not exceedingly restrictive.

3 Popularity-driven Caching Scheme in Meta-search Engines

Popular queries are ones that are presented repeatedly to web search engines by users, and they represent a significant portion of all submitted queries. Popular queries tend to reflect the public interests or trends in Internet searching at a given time. For example, searching expressions about North Korea were heavily posed to search engines in Korea when the historical summit meeting between two Korea's was taking place in June 2000. Generally speaking, popular queries depend on significant events, such as political, social, cultural events; these are sensitive to time and changes over time.

For a query consisting of a single word, we define the notion of popularity, which represents popularity of a query among all queries. The popularity should be normalized on the basis of posting cycles, the number of lists of popular queries, and the predefined maximum number of popular queries in the lists. The values that the

popularity can take range from zero to one; the higher value means more popular. Words with zero popularity are ones that never occur in the popular query lists under consideration. Three different popularities are defined, namely *unit popularity* (UP), *accumulated popularity* (AP), and *fused popularity* (FP).

Before we present how to calculate popularities, we need to define one variable W. W is to be set in advance, and it will be used to normalize popularities. The only restriction on W is that it be greater than the expected maximum number of popular queries at all times. W is independent of particular search engines. Once W is set in the beginning, W remains unchanged permanently.

UP is calculated on a single list of popular words. UP is calculated on every posting of popular queries for each search engine. Let $R_i^j(X)$ be the rank of popular query X in the ith search engine's j^{th} posting, $UP_i^j(X)$ denote UP of popular query X in the ith search engine's j^{th} posting. Here is the formula of UP:

$$UP_i^j(X) = \frac{W - R_i^j(X) + 1}{W}$$

$R_i^j(X)$ is always equal to or greater than one. UP is essentially a normalized value of rank of a query in a single list of popular queries. Note that $0 < UP_i^j(X) \leq 1$. If a query is listed at the same rank in the two different lists of popular queries, then the UPs of the query are the same regardless of the number of popular queries in the lists.

AP is obtained by accumulating UP's of the same search engine over a predetermined period of time. The number of accumulations within a time period is called the order. We use AP's to deal with different search engines that post popular query lists in different posting cycles (for example, daily and weekly). AP gives popularities of queries that shall be collected over the same period of time. Let n_i denote the order of AP of the ith search engine, $AP_i(X)$ denote the AP of popular query X in the ith search engine. Then, $AP_i(X)$ is calculated as below:

$$AP_i(X) = \frac{1}{n_i}\sum_{j=1}^{n_i} UP_i^j(X)$$

Since AP is a summation of UP divided by the number of the order, we have $0 < AP_i(X) \leq 1$. A query gets a high value of $AP_i(X)$ if the query X appears often in popular query lists of the ith search engine. If a query appears only once in popular lists, then AP of the query becomes low (precisely speaking, UP is the same as the division of AP by the order).

FP presents integrated popularity collected by different search engines that post popular query lists periodically. The number of search engines in consideration normalizes FP, which is used in our cache scheme. Let N be the number of search engines in consideration. FP is defined as below:

$$FP(X) = \max\left(0, \frac{1}{N}\sum_{i=1}^{N} AP_i(X)\right)$$

All queries, whether they are popular or not, have FP's. A query has a zero value of FP if it does not appear in any of popular query lists.

In our computation, a query gets higher popularity if: (1) it has higher ranking in a popular query list, (2) it appears often in many query lists collected by different search engines over the same period of time or, (3) it appears often in many query lists collected over different time periods by the same search engine. It should be

noted that our computation is flexible enough to accommodate any number of query lists that are posted over any period of time by any number of different search engines.

Example 1. Consider two web search engines. Let W be 200. The first engine posts 100 popular queries weekly, and the second posts 50 popular queries daily. The order of AP (n_1) of the first is assigned to be 1, and the order (n_2) of the second is 7.

Now assume that a query "Napster" is listed as follows. In the first search engine, "Napster" has the 2nd rank. In the second search engine, the word is listed in the 15th, 10th, 5th, 1st, 8th, 9th, and 12th rank, respectively in the daily popular query lists. In the first search engine, the accumulated popularity of "Napster" equals the unit popularity, since the order of accumulation is one.

$$AP_1(Napster) = UP_1^1(Napster) = \frac{200-2+1}{200} = 0.995$$

In the second search engine, the accumulated popularity of "Napster" is computed as:

$$AP_2(Napster) = \frac{1}{7}\sum_{j=1}^{7} UP_2^j(Napster)$$

$$= \frac{1}{7}\left(\frac{(200-15+1)+(200-10+1)+(200-5+1)+(200-1+1)+(200-8+1)+(200-9+1)+(200-12+1)}{200}\right) = 0.962$$

Then, FP(Napster), the fused popularity of "Napster", is computed as:

$$FP(Napster) = \frac{1}{2}\sum_{i=1}^{2} AP\ (Napster) = \frac{1}{2}\left(AP_1(Napster) + AP_2(Napster)\right) = 0.9785$$

The fused popularity of "Napster" is then 0.9785. ■

Now we present a popularity-driven replacement (PDR) algorithm. PDR utilizes both FP and reference counters of a query in the cache replacement process. The reference counter of a query holds the number of references (accesses) of the query stored in the cache.

The cache in consideration consists of a number of slots. Each slot has five components (X, FP, C, T, R), where X denotes a query presented by the user, FP denotes the fused popularity of the query X, C denotes the current reference counter of the query X, T denotes the time in which a search result is constructed, and R denotes a search result (here, a collection of HTML pages).

Search engines are known to update their data from time-to-time, perhaps monthly, quarterly, etc. Since we do not want to return stale data to users, we keep track of the time at which search results are collected. For each search engine, we maintain a variable for this purpose. This variable indicates how often a search engine updates its data, and it is used to determine staleness of cached data. If data is stale, we discard cached data and connect to the search engines to receive fresh data. The newly received data will be stored in the slot, and the reference counter of the slot is incremented by one.

Popularity, which is the primary criterion for cache replacement, changes over time. Fused popularities should be updated if necessary. We introduce the notion of *popularity update cycle*. During popularity update cycle, we re-calculate FP's of all queries using new lists of popular queries, and FP's of all queries stored in the cache are updated accordingly. A non-popular query (i.e., a query with zero popularity) in the cache could become a popular query and vice versa on popularity update cycle.

However, the reference counter in the slots remains unchanged on popularity update cycle.

Figure 2 shows the pseudo code of PDR. The input of the algorithm is a query X, and the output is a search result (R). When requested data is already cached, we then increment the reference counter by one and return the search result.

```
Function Popularity-base replacement()
  Let EVICT be the slot number such that FP is minimum and not zero;
  If (FP of slot EVICT ≥ FP(X)) then
      Return R and exit;
  End if
  Flush slot EVICT into disk;
  Store (X, FP, 1, T, R) into slot EVICT and return R and exit;
End Function
Function Reference-base replacement()
  Let EVICT be the slot number such that C is minimum and FP is zero;
  Flush slot EVICT into disk;
  Store (X, FP, 1, T, R) into slot EVICT and return R and exit;
End function

If X is already in the cache (say slot i) then
  Fetch slot i in the cache;
  If slot i is not stale then
      Increment C by one;
      Return R of slot i and exit;
  End if
  Construct the search result (R) of X by accessing search engines;
  Increment C by one;
  Store (X, FP, C, T, R) into slot i and return R and exit;
End if
Construct the search result (R) of X by accessing search engines;
If cache is not full then
  If FP(X) is not zero then
      If ((1 - N_P/N_S) > O_NP)
        /* The ratio of non-popular queries is greater than threshold */
        Get the next available slot (let the slot number be FREE);
        Store (X, FP, 1, T, R) into slot FREE and return R and exit;
      End if
      /* The ratio of non-popular queries is no greater than threshold */
      Popularity-base replacement();
  End if
  /* When X is a non-popular query */
  Get the next available slot (let slot number be FREE);
  Store (X, FP, 1, T, R) into slot FREE and return R and exit;
End if
/* When the cache is full */
If FP(X) is not zero then
  If ((1 - N_P/N_S) > O_NP) then
      /* The ratio of non-popular queries is greater than threshold */
      Reference-base replacement();
  End if
  /* The ratio of non-popular queries is no greater than threshold */
  Popularity-base replacement();
End if
/* When X is a non-popular query */
Reference-base replacement()
```

Fig. 2. The PDR Algorithm

PDR reserves a portion of slots for non-popular queries at all times. Let N_S denote the number of cache slots, N_P denote the current number of popular queries in the cache. Then N_P/N_S denotes the ratio of popular queries over all the queries stored in cache and $1 - N_P/N_S$ denotes the ratio of non-popular queries over all the stored queries. PDR guarantees $(1 - N_P/N_S)$ to be greater than a threshold at all times, i.e., $(1 - N_P/N_S) > O_{NP}$, where O_{NP} is the minimum ratio of non-popular queries over all the loaded queries. O_{NP} is to be set in advance.

Even though there are free slots in the cache, PDR applies the popularity-based replacement if a newcomer is a popular query and the ratio of non-popular queries is less than the threshold. Such replacement is necessary in order to reserve slots for non-popular queries coming later.

In order to achieve this, PDR makes use of two different criteria (i.e., popularities and reference counters) for cache replacement. There are two cases in which the first criterion (popularities) is applied; first, when cache is full, a newcomer has non-zero popularity (a popular query) and the ratio of non-popular queries is no greater than the threshold (O_{NP}), second, when there are free slots in the cache, a newcomer has non-zero popularity (a popular query) and the ratio of non-popular queries is no greater than the threshold. In those cases, a newcomer replaces the slot having the minimum value of popularities, as long as the popularity of the newcomer is bigger than the minimum value of popularities.

The second criterion (reference counters) is applied only when the cache is full. The first case is when a newcomer has zero popularity (a non-popular query), the second case is when a newcomer has non-zero popularity and the ratio of non-popular queries is greater than the threshold. In both cases, a newcomer replaces the slot that has zero popularity and the minimum value of reference counters.

4 Experimental Evaluation

Experimental evaluation of our caching scheme has been carried out. We implemented and evaluated LRU, LFU, and PDR algorithms with Java development kit version 1.3 (JDK1.3). The experiments were performed on a Pentium III-650 machine with 256 Mbytes main memory.

For the experiments, we used real data. We selected three search engines that are posting a reasonable number of popular queries periodically. Two of them are posting popular queries weekly, and one search engine is posting daily. As for users' input queries, we used a log file that was generated by a commercial search engine in Korea. It contained 781,550 queries totally.

In order to see the effect of accumulation of popular queries, we made three different lists of popular queries, each of which was collected over one week (144 popular queries), two weeks (178 popular queries), and four weeks (194 popular queries), respectively. We denote them as PDR-1, PDR-2, and PDR-4, where the suffixes 1, 2, 4 denote the number of weeks. Duplicate terms are always purged in the lists. There is much commonality in the lists posted by search engines, as expected.

The metric we use is a cache hit ratio, which is a ratio of the number of hitting queries over the number of total single queries submitted. The second metric is the performance gain ratio of LFU and PDR over LRU. LRU is the most widely used one in practice, so it can serve as a reference point of performance. The gain ratio of LFU or PDR is calculated by dividing the hit ratios of LFU or PDR over the hit ratio of LRU, and it is presented by percentage.

Our evaluation has been focused on three aspects of the PDR algorithm: performance effect of cache size, the changing of the minimum ratio of non-popular queries in cache, and accumulation of popular queries. For the first evaluation, we

have varied the number of slots in the cache. We tested seven cases, with the number of slots 50, 75, 100, 125, 150, 180, and 200, respectively. Figure 3 shows the hit ratios of LRU, LFU and PDR-1. As the number of slots increases, the hit ratio also increases. PDR performed the best in all cases except in the 50-slot case, where LFU was the best. PDR exhibits 6-8% of improvement to LRU in terms of the hit ratio in all cases. W was assigned to be 300 and O_{NP} was 0%.

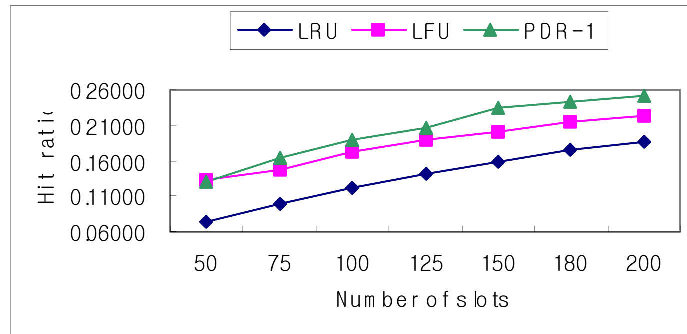

Fig. 3. The Hit Ratios of LRU, LFU, and PDR-1

Figure 4 shows the gain ratios of LFU and PDR-1 over LRU. It is interesting to note that the gain ratios of LFU and PDR decrease as the number of slots increases. But LFU decreases more rapidly than PDR does. In case of the large cache, the performance gain of LFU over LRU becomes marginal, which implies that LRU becomes to work well. One lesson we learned is that the cache size is a critical performance factor in cache replacement schemes.

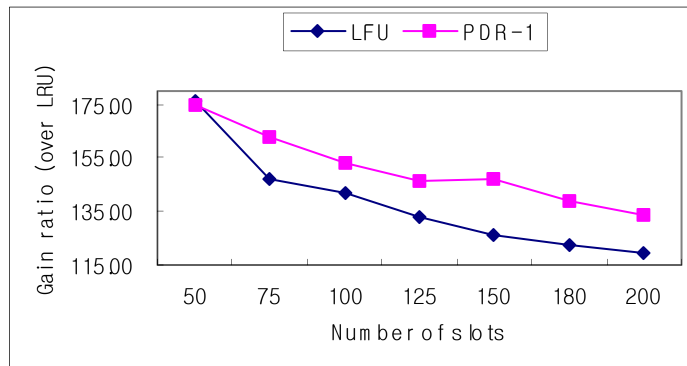

Fig. 4. The Gain Ratios of LFU and PDR-1

Second, we have varied the minimum ratio of non-popular queries in cache. We tested six cases, with 0%, 5%, 10%, 15%, 20%, and 25%, respectively. Figure 5 shows the gain ratios of PDR-1.

Except in the 50-slot case, PDR performs the best in all cases when O_{NP} uses 5%. In almost every case, the gain ratio decreases as O_{NP} increases over 5%. This result implies that loading the cache with too many non-popular queries does not help to

increase performance. Note that the gain ratios of 200 slots are the same regardless of O_{NP}. This is because the number of slots is greater than the number of popular queries. Specifically, all 144 popular queries are eventually loaded in the cache, and non-popular queries compete to be loaded in the 56 remaining slots. This phenomenon happened partially when the numbers of slots were 150 and 180.

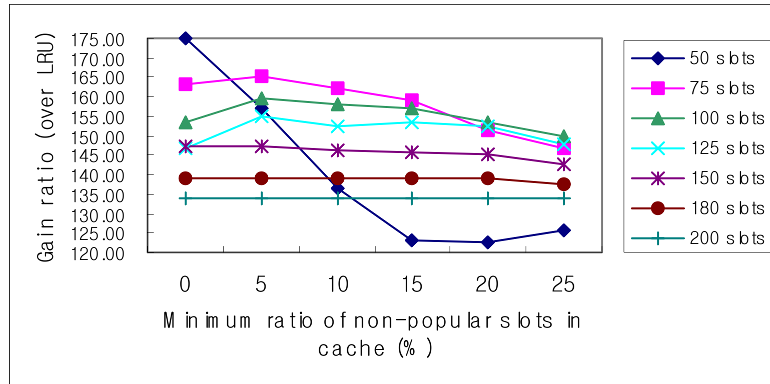

Fig. 5. The Gain Ratio of PDR-1 over LRU

Third, we evaluated the effect of the accumulation of popular queries by running the three fused popularities. Figure 6 shows the hit ratios of PDR-1, PDR-2 and PDR-4. Each accumulation shows similar behavior. When there are some cases (say 50-slot and 200-slot), PDR-2 and PDR-4 beat PDR-1 consistently. This result implies that accumulation of popular queries over a long period of time (say over one month) does not help increase the hit ratio. The accumulation of popular queries over two weeks or one week would be a reasonable choice in terms of caching performance.

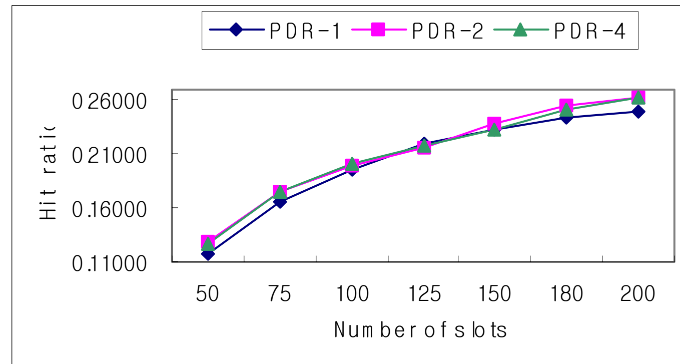

Fig. 6. The Hit Ratios of PDR-1, PDR-2, and PDR-4

With three aspects of performance evaluation, we come to a conclusion as follows: (1) the page replacement policy utilizing both popularities and reference counters of queries works out well, (2) the cache size is a critical performance factor in cache replacement schemes, (3) reserving a portion of slots for non-popular queries help

increase performance (in our experiments, it was 5%), (4) accumulation of popular queries does not help.

5 Closing Remarks

We have presented a popularity-driven caching scheme for meta-search engines. Computation of query popularity is given, and the PDR algorithm for cache replacement is described. Empirical evaluations and performance comparisons with LRU and LFU have been carried out on real data. PDR exhibits satisfactory behavior in almost all cases.

On posting popular queries, some search engines have an internal policy that censors indecent queries from their lists of popular queries. A commonly exercised policy in the lists is to exclude pornographic words, which otherwise would appear in the lists as very popular words. The experimental results shown in this paper are therefore conservative, considering that search engines self-censor their lists of popular queries. If we had used unfiltered lists of popular queries for experiments, then PDR would show even better performance than is described here.

Our future work includes expansion of the popularity calculation to consider popular queries that could be collected internally. When EzFinder is operational, we can collect user queries. Such internal popularities can be merged with external ones. If we had accumulated the “indecent” queries on our own list, then we could actually measure the true improvement. This is a topic of future research.

References

1. J. Cheong, and S. Lee: A Boolean Query Processing with a Result Cache in Mediator Systems, Proceedings of Advances in Digital Libraries, IEEE (2000), 218-227
2. B. Chidlovskii, C. Roncancio, and M. Schneider: Semantic Cache Mechanism for Heterogeneous Web Querying, WWW8 (1999), 1347-1360
3. S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan: Semantic Data Caching and Replacement, Proceedings of the 22nd VLDB Conference (1996), 330-341
4. T. Johnson, and D. Shasha: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, Proceedings of the 20th International Conference on VLDB (1994), 439-450
5. S. Lawrence, and C. Giles: Accessibility of Information on the Web, Nature, 400 (1999), 107-109
6. D. Lee, and W. Chu: Semantic Caching via Query Matching for Web Sources, Proceedings of the 8th International Conference on Information Knowledge Management (1999), 77-85
7. E. O’Neil, P. O’Neil, and G. Weikum: The LRU-K Page Replacement Algorithm for Database Disk Buffering, Proceedings of the ACM SIGMOD (1993), 297-306
8. J. Robinson, and M. Devarakonda: Data Cache Management Using Frequency-Based Replacement, Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1990), 134-142
9. Scime, and L. Kerschberg: WebSifter: An Ontology-based Personalizable Search Agent for the Web, Proceedings of the International Conference on Digital Libraries, Research and Practice (2000), 439-446