

# Admission Policies for Caches of Search Engine Results

Ricardo Baeza-Yates<sup>1</sup>, Flavio Junqueira<sup>1</sup>, Vassilis Plachouras<sup>1</sup> and  
Hans Friedrich Witschel<sup>2</sup>

<sup>1</sup> Yahoo! Research, Barcelona, Spain

<sup>2</sup> University of Leipzig, Germany

{rby, fpj, vassilis}@yahoo-inc.com,  
witschel@informatik.uni-leipzig.de

**Abstract.** This paper studies the impact of the tail of the query distribution on caches of Web search engines, and proposes a technique for achieving higher hit ratios compared to traditional heuristics such as LRU. The main problem we solve is the one of identifying infrequent queries, which cause a reduction on hit ratio because caching them often does not lead to hits. To mitigate this problem, we introduce a cache management policy that employs an admission policy to prevent infrequent queries from taking space of more frequent queries in the cache. The admission policy uses either stateless features, which depend only on the query, or stateful features based on usage information. The proposed management policy is more general than existing policies for caching of search engine results, and it is fully dynamic. The evaluation results on two different query logs show that our policy achieves higher hit ratios when compared to previously proposed cache management policies.

## 1 Introduction

Without search engines, finding new content on the Web is virtually impossible. Thus, a large number of users submit queries to search engines on a regular basis in search of content. As the number of users is large and the volume of data involved in processing a user query is high, it is necessary to design efficient mechanisms that enable engines to respond fast to as many queries as possible. An important mechanism of this kind is caching. In search engines, users submitting popular queries can benefit from a mechanism that stores the results for such queries in a cache memory, as the engine does not need to recompute results that are requested frequently enough. Caching query results then improves efficiency if the cached queries occur in the near future.

A cache comprises a memory space and an implementation of a cache management policy. As cache memories are limited in size, it is necessary to evict entries when the cache is full and there is a new entry to add. To evict entries, a cache has to implement an *eviction policy*. Such a policy ideally evicts entries that are unlikely to be a *hit*, *i.e.*, to be requested while in the cache. A simple and popular strategy is to evict the least recently used item from the cache [1]. This policy is known as LRU (Least Recently Used). In search engines, the entries in a cache can comprise, for example, query results and posting lists [2]. In this paper, we focus on query results. Henceforth, we say that a query is cached to denote that the results of the query are cached.

The problem with using only eviction policies in Web search engines is that the results for *all* queries are admitted to the cache, including those that will never appear again. Storing the results for these queries turns out to be fruitless since they cannot be cache hits. Until they are evicted from the cache, they “pollute” it in the sense that they use cache space, but they do not generate any cache hit. Since the frequency of queries follows a power law, there is a great percentage of queries that never appear again – at least not in a near future.

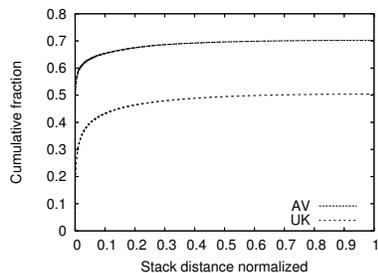
In this work, we propose the use of *admission policies* to prevent infrequent or even singleton queries from polluting the cache. Such policies are implemented in the form of an estimator that predicts whether a query is infrequent or whether it is frequent enough to be cached. The estimator can use either stateless features, which depend on each query, or stateful features, which are computed from usage information. We assume that there is a relation between the features and the future frequency of queries, even though we do not explicitly model it. To evaluate the efficiency of our policies, we use the hit ratio of the cache, instead of accuracy of frequency prediction.

Our cache management policy for search engine results comprises an eviction policy and an admission policy, and it divides the memory allocated for caching into two parts. The queries cached in the first part are the ones that the admission policy predicts as frequent or, more generally, likely to be a hit in the future. The remainder of the queries are cached in the second part of the memory. The experimental results from applying an admission policy, as described above, show that we obtain improvements over LRU and SDC, which is currently one of the best management policies for caching results in Web search engines.

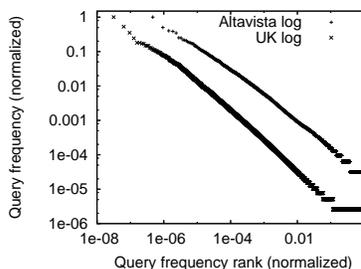
*Related work.* The observation on filtering out infrequent queries has not been directly used in the design of caches for Web search engines so far. Markatos [3] investigated the effectiveness of caching for Web search engines. The reported results suggested that there are important efficiency benefits from using caches, due to the temporal locality in the query stream. Xie and O’Hallaron [4] also found that the distribution of query frequencies follows a Zipf distribution, very popular queries are issued by different users, and longer queries are less likely to be shared by many users.

On cache management policies, Lempel and Moran [5] proposed one that considers the probability distribution over all queries submitted by the users of a search engine. Fagni *et al.* [6] described a Static Dynamic Cache (SDC), where part of the cache is *read-only* or *static*, and it comprises a set of frequent queries from a past query log. The dynamic part is used for caching the queries that are not in the static part. This last work is the most closely related to the problem of polluting queries, because it actively protects the most frequent queries and always keeps them cached. The solution of SDC, however, only addresses the problem of polluting queries indirectly, while it introduces the constraint of the static cache.

In a different context, on caching of memory pages in operating systems, there has been work on splitting the memory available for caching according to the type of traffic. For example, the adaptive replacement cache (ARC) [7] uses two lists, one for recent references and one for frequent references. One the main features of ARC is that it adapts the lengths of the two lists. Compared to ARC, we instead try to separate very frequent references from infrequent ones, and we investigate different policies.



**Fig. 1.** Measure of temporal locality.



**Fig. 2.** Query frequency distributions.

Using multiple levels of caches for storing query results and postings of query terms [8] or even intersections of posting lists for query terms [9] result in improved hit ratio values (the hit ratio is the total number of hits over the total query volume). In this work we focus on caching of query results and we do not consider the problem of caching posting lists.

Finally, caching has been considered for Web applications that generate content dynamically [10, 11]. One main difference from our work to the work of Sivasubramanian *et al.* and Olston *et al.* is their focus on applications that access one or more databases. One of the main challenges is to maintain database consistency as transactions change the state of the backend database. Search does not have a similar constraint because queries do not change the state of the data structures used to compute results. For federated databases, Malik *et al.* [12] have used admission policies for caching to minimize network communication overhead. Although they use the same concept of admission policy, the policies they use are different compared to the ones we propose.

## 2 Data Characterization

We use two query logs in our experiments. The first one corresponds to queries submitted to the Altavista Web search engine during a week in autumn 2001. The second query log corresponds to queries submitted to `yahoo.co.uk` during one year.

The queries in both logs have very similar length characteristics. The average length in characters and words of queries in the Altavista log are 17 and 2.6, respectively. The queries in the UK log consist on average of 17 characters and 2.5 words.

For caching, temporal locality is an essential property because if consecutive occurrences of the same query happen close together in time, then eviction policies can be highly efficient. To measure and compare temporal locality of the two query logs, we use a traditional technique that consists of computing the stack distance between two occurrences of the same query [13]. The stack abstraction works as follows. When processing a query stream, we process one query at a time. For each new query  $q$ , if  $q$  is not in the stack, then we push it onto the stack, and count it as an infinite stack distance.

Otherwise, suppose that  $d$  is the depth of the query  $q$  in the stack, remove  $q$  from its current place in the stack and push it onto the top. The stack distance in this case is  $d$ .

Figure 1 shows the cumulative stack distance distribution for normalized distance values of both query logs. We use the total volume of each log to normalize the stack distance of the corresponding log. Note, however, that the total volume of the Altavista log is smaller compared to the total volume of the UK log. Thus, the same normalized value corresponds to a smaller absolute value in the Altavista log.

In this graph, there are two important observations. First, the highest probability is 0.7 for the Altavista log and 0.5 for the UK log. These probability values are not 1.0 because of singleton queries and compulsory misses. For such queries, the stack distance is infinite. Thus, in the Altavista log, we observe that a larger fraction of the query volume comprises repeated queries. Second, for the same distance value, the cumulative probability is higher for the Altavista log. If we pick  $x = 0.1$  as an example, then the difference between the cumulative probabilities is roughly 0.2. Note that the difference between the two curves is roughly the same across the values in the  $x$  range. It happens because the difference for the maximum value of  $x$  is 0.2 and the increments are small for values of  $x$  larger than 0.1 due to the small fraction of large distance values.

As the stack distance between two consecutive occurrences of the same query has a higher probability of being short for the Altavista log, we conclude that the Altavista log presents significantly more temporal locality compared to the UK log. This is not surprising because of the higher volume of the UK log and its time span.

The actual frequency distributions of queries of the two logs, shown in Figure 2, confirm the conclusions above. From the figure, both distributions follow a power law distribution: the distribution for the Altavista log has slope  $-1.71$  and the distribution for the UK log has slope  $-1.84$ . For the Altavista log, the singleton queries, which appear only once, correspond to 19% of the total query volume and to 62% of the unique queries. A cache that has an infinite amount of memory to store the results for all observed queries without any eviction, would achieve a hit ratio of 70.21%. For the UK query log, the singleton queries correspond to a higher percentage of the total volume of queries. More specifically, the set of singleton queries comprise 44% of the total volume and 88% of the unique queries. An infinite cache achieves 50.41% hit ratio. In the next section, we elaborate on the impact of the infrequent queries on caching.

### 3 Polluting Queries

A predominant characteristic of query streams is the presence of infrequent queries, in particular of queries that appear just once. For such queries, caching is often not effective because either there is a large number of other queries separating consecutive occurrences, or simply they never occur again. Moreover, using cache space for such a query might imply evicting the results of another more frequent query, thus increasing the number of cache misses. The basic idea of our approach consists of determining which queries are infrequent ones, and caching them in a separate part of the memory.

Table 1 illustrates the benefits of our approach. We compare two different caching policies: the optimal-admission policy and the least recently used (LRU) policy. The

optimal-admission policy knows when a query will never appear again, and it does not cache such a query. On the other hand, if a query will appear again in the future, then the policy accepts it and it uses the LRU policy to determine which query to evict, assuming the cache is already full.

From the table, we observe that for the Altavista log the LRU policy achieves a high hit ratio already (between 59% and 65%), but the optimal-admission policy is still able to obtain even higher values (between 67% and 70%). For the UK log, however, using the optimal-admission policy enables an absolute increase in the hit ratio of over 10% in all three cases. This difference is mainly due to the amount of temporal locality in these logs, observed in the previous section. As the UK log presents less temporal locality, the policy controlling admission has a higher impact.

**Table 1.** Hit-ratio (%) comparison between the optimal-admission policy and LRU for different cache sizes in number of queries.

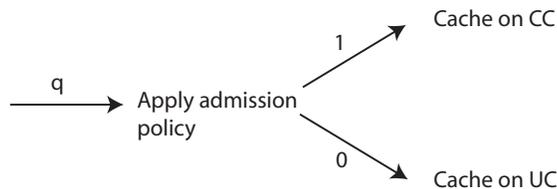
Cache size	Optimal		LRU	
	AV	UK	AV	UK
50k	67.49	32.46	59.97	17.58
100k	69.23	36.36	62.24	21.08
250k	70.21	41.34	65.14	26.65

These results show that if we design heuristics that accurately determine which queries do not occur frequently, then we can improve significantly the hit ratio of caches for query results.

## 4 Admission Policies and Caching

This section describes a family of cache management policies (AC) that use an admission policy to separate infrequent queries from frequent ones. There are different ways of making a module implementing an admission policy interact with the cache. For example, we can have each query  $q$  sequentially evaluated: the admission policy first evaluates  $q$ , and, depending on the outcome, the cache processes the query in different ways. Alternatively, the admission policy evaluates the query and the cache verifies if the results are stored in parallel. The cache, however, has to wait for the outcome of the admission policy evaluation before applying its eviction policy. In this paper, we use the former, as the latter is mainly an optimization.

The proposed cache has two fully-dynamic parts. The first part is an admission-controlled cache. We call this part *controlled cache* (CC), because it only admits those queries that the admission policy classifies as future cache hits. All queries the admission policy rejects are admitted to the second part of the cache, which we call *uncontrolled cache* (UC). In our experiments in the next section, the uncontrolled cache implements a regular cache, more specifically LRU. We also use LRU for the controlled cache. Figure 3 shows how a query stream is processed with AC.



**Fig. 3.** Sequence of actions performed when a new query  $q$  arrives at a cache.

The rationale behind this management policy is similar to that of SDC [6], but it makes the first part of the cache more flexible and dynamic rather than static: the controlled part will contain those queries that are likely to be hits. Ideally, with a perfect admission policy, there would be no need for the uncontrolled cache. However, implementing such an admission control policy is in general difficult. More specifically, if this admission control policy is based, for example, on frequency, then its estimation on hits and misses is not perfect due to the temporal locality of some infrequent queries.

The uncontrolled cache can therefore handle queries that are infrequent, but appear in short bursts. Recall that the admission policy will reject queries that it determines to be infrequent. Infrequent (or unpopular) queries then may be asked again by the same user and within a short period of time. The uncontrolled cache handles these cases. Thus, AC guarantees that fewer infrequent queries enter the controlled cache, which is expected to handle temporal locality better.

To decide upon which queries to cache in the controlled part and the ones to cache in the uncontrolled part, we use an admission policy. Given a stream of queries  $Q_s$ , an admission policy is a function  $f : Q_s \rightarrow \{0, 1\}$  that decides, for each query  $q \in Q_s$  of the stream, whether it should be cached in the controlled part ( $f(q) = 1$ ) or in the uncontrolled part ( $f(q) = 0$ ). The generality of AC lies in the fact that various admission policies can be applied. For example, SDC is now a special case, which results from the following admission policy:

$$f(q) = \begin{cases} 1 & \text{if } q \text{ is among the } |CC| \text{ most frequent queries in a given training set} \\ 0 & \text{else} \end{cases} \quad (1)$$

With this admission policy, CC is static (no query ever needs to be evicted) and after some time, it will contain the same queries that would be in the static part of an SDC cache.

Now, other admission policies – that admit more queries – can be used to make CC more dynamic and hence more effective. To design a good admission policy, one needs to think of *features* that may be useful in distinguishing future cache hits from future misses. A feature in this context is some property of a query that the admission policy uses to determine in which part of the memory to cache it. Some options for such features will be discussed in the next sections.

## 4.1 Stateful Features

Stateful features are based on historical usage information of a search engine, and in general, they require extra memory space to hold statistics. Typically, these statistics are related to the frequency of query substrings (*e.g.*, words, *n-grams*) or the whole query. For example, a simple admission policy with a “stateful” feature admits all queries whose frequency in a training set of past queries is above a threshold  $k$ . It may be necessary to tune this threshold before applying the policy. In addition, the frequency statistics may need to be updated in regular intervals.

With respect to the extra amount of memory needed for the frequency-based feature, there are three important observations. First, this amount of memory is small compared to the total cache memory. For example, if the cache stores 20Kbytes for 100K queries, then it requires approximately 2GB of memory, while the amount of memory required for the state of the Altavista query log, corresponding to 1.4 million queries, is approximately 30MB, which is less than 2% of the space used for the cache. Second, we did not try to make the data structure holding frequency information efficient with respect to space. In fact, the 30MB value includes all the queries in the training set, although we do not need to keep all of them depending on the value of the frequency threshold. Alternatively, one can use a more space-efficient representation for queries, such as Bloom filters or other hash-based schemes. Hence, the 30MB value is an upper bound in the case of the Altavista query log. Even considering this additional amount of memory for LRU and SDC, preliminary experiments have shown that the resulting increase in hit ratio is negligible, as we have verified. Third, search engines often maintain statistics about the query stream for reasons such as improving the quality of the results returned. In this case, the information the feature requires may be readily available, and no extra amount of memory is necessary. For these reasons, we decided not to consider this extra amount of memory for LRU and SDC in our experiments.

## 4.2 Stateless Features

A stateless feature is a feature that can be readily computed from the query stream itself, without making use of collected information. The advantage of stateless features is that they neither require keeping track of statistics (and hence no update over time) nor memory space for storing such information.

Examples of potentially useful stateless features include the length of a query (in characters or words) or the number of non-alphanumeric characters in the query. The idea behind these features is that long queries or those containing many non-alphanumeric characters have lower probability of being popular. The corresponding admission policies require a threshold  $k$ . Polluting queries are then queries longer than  $k$  words or characters, or containing more than  $k$  non-alphanumeric characters.

## 5 Evaluation

In this section, we evaluate AC. We describe in detail the experimental setting, and then we present our results.

## 5.1 Experimental Setting

To evaluate the performance of AC, we conduct a number of experiments using the logs described in Section 2. More precisely, we use three different admission policies and compare them to an LRU baseline cache without admission policy and an SDC cache. In SDC, we have a hit when a query exists in either its static or its dynamic part. In the case of a miss, the query is introduced in the dynamic part. Here, the dynamic part of SDC implements LRU.

We divide both logs into a training set of queries  $M$ , which we use to train an admission policy, and a test set  $T$ . The training set consists of the same absolute number (namely 4.8 million) of queries in both cases. For our experiments, such a number of queries is sufficient to obtain reliable frequency estimates and a test set that is large enough for experiments with the Altavista log. Fagni *et al.* use the same two thirds/one third split for the Altavista log [6]. We do not consider pages of results in our experiments. That is, we consider two queries requesting different pages of results as the same query. The underlying assumption is that it is not expensive to cache all (or at least enough) results for each query in order to satisfy all these different requests.

The performance of a cache was measured by computing the hit ratio of a cache simulator on the test set  $T$ . Since SDC needs to be started with a warm cache, all caches start the evaluation phase warm. For SDC, we warm (cf. [6], section 5) its static part by populating it with the most frequent queries from  $M$ , and we warm its dynamic part by submitting the remaining queries to its dynamic part. For LRU and AC, we run the policy on the training set to warm their respective caches. In all cases, however, we count hits only for the test set  $T$ . We also report the hit ratio of the test set of an infinite cache warmed with the corresponding training set. Note that the hit ratio values for an infinite cache are different from the ones we report in Section 2 due to the split between test and training.

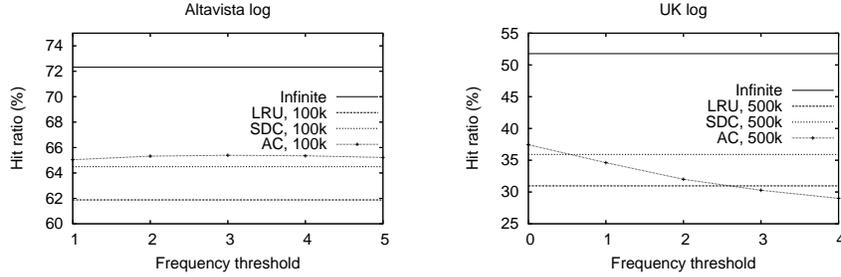
For the Altavista log, we use caches of size 50K and 100K and for the UK log of size 100K and 500K. For AC and SDC, the ratio of the controlled vs. uncontrolled (or static vs. dynamic) parts varies according to the parameters of the experiment. This is because the ratio that gives the highest hit ratio depends on the cache size and on properties of the query log.

All admission policies that we experiment with consist of just one feature and a corresponding threshold. Among these, there is one stateful feature, namely the frequency of the query in  $M$  ( $PastF$ ), and two stateless features, namely the length of the query in characters ( $LenC$ ) and words ( $LenW$ ).

## 5.2 Results

Here we present the evaluation results for our proposed admission-controlled dynamic caches, using both the stateful and the stateless features. In each table, we present the results from the best split between static and dynamic cache for SDC, and controlled and uncontrolled cache for AC.

We start with the stateful feature  $PastF$ . The results are shown in Table 2 and Figure 4. When we use the past frequency to decide whether to admit queries, we obtain improved hit ratios over SDC. We can see that small caches require higher frequency



**Fig. 4.** Hit ratios of the stateful feature for the Altavista and UK query logs.

thresholds because small caches can only effectively store queries that appear frequently enough so that they are not evicted. In the case of the AC cache with a capacity of 50K on the Altavista log, we can see that the best thresholds  $k_f$  are 7 and 8, where  $k_f$  is the frequency threshold.

**Table 2.** Hit ratios (%) for the Altavista and the UK query logs using AC, where a query is admitted in the controlled part of the cache if its frequency in the past is greater than  $k_f$ . The last row shows the percent of the LRU hit ratio relative to the hit ratio of an infinite cache.

	AV		AV		UK
Infinite	72.32	Infinite	72.32	Infinite	51.78
Sizes	50K	Sizes	100K	Sizes	100K 500K
LRU	59.49	LRU	61.88	LRU	21.03 30.96
SDC	62.25	SDC	64.49	SDC	29.61 35.91
AC $k_f = 6$	63.16	AC $k_f = 1$	65.04	AC $k_f = 0$	28.55 <b>37.45</b>
AC $k_f = 7$	63.19	AC $k_f = 2$	65.32	AC $k_f = 1$	29.94 34.62
AC $k_f = 8$	<b>63.19</b>	AC $k_f = 3$	<b>65.39</b>	AC $k_f = 2$	<b>30.28</b> 32.00
AC $k_f = 9$	63.16	AC $k_f = 4$	65.35	AC $k_f = 3$	29.32 30.28
AC $k_f = 10$	63.08	AC $k_f = 5$	65.22	AC $k_f = 4$	27.94 29.00
LRU/Infinite	82.26	LRU/Infinite	85.56	LRU/Infinite	40.61 59.79

Next we consider the stateless features. Table 3 presents the results when the AC cache uses the length in characters or words of the query, to predict whether a query is worth caching. In all cases, AC outperforms baseline LRU. The threshold that resulted in the best performance is  $k_c = 20$ , except for the case of the Altavista log with an AC cache of 50K. The results are similar when we use the length in words of queries, and the best threshold value is  $k_w = 2$ . Figure 5 shows the resulting hit ratios for the two features,  $LenC$  and  $LenW$ , for an AC cache of 500K used with the UK log.

Compared to LRU, AC achieves a higher performance because it is able to filter out some queries that pollute the cache. However, the stateless features are not as efficient as a predictor based on frequency, and consequently they do not outperform SDC. In the next section, we discuss further the advantages of using such features, as well as other possibilities for features that may improve performance.

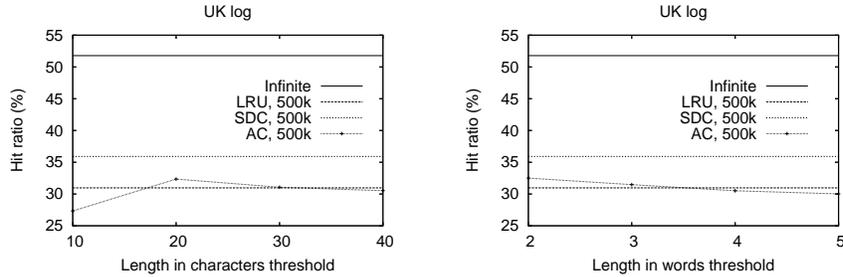
**Table 3.** Hit ratios (%) for the Altavista and the UK query logs using AC, where a query is not admitted to the controlled part of the cache if its length in characters (words) is greater than  $k_c$  ( $k_w$ ). The last row shows the percent of the LRU hit ratio relative to the hit ratio of an infinite cache.

	AV		UK	
Infinite	72.32		51.78	
Sizes	50K	100K	100K	500K
LRU	59.49	61.88	21.03	30.96
SDC	<b>62.25</b>	<b>64.49</b>	<b>29.61</b>	<b>35.91</b>
AC $k_c = 10$	60.01	59.53	17.07	27.33
AC $k_c = 20$	58.05	62.36	22.85	32.35
AC $k_c = 30$	56.73	61.91	21.60	31.06
AC $k_c = 40$	56.39	61.68	21.19	30.53
AC $k_w = 2$	59.92	62.33	23.10	32.50
AC $k_w = 3$	59.55	61.96	21.94	31.47
AC $k_w = 4$	59.18	61.60	21.16	30.51
AC $k_w = 5$	59.01	61.43	20.81	30.02
LRU/Infinite	82.26	85.56	40.61	59.79

## 6 On the Design of New Features

Table 1 shows that identifying perfectly if there will be another occurrence of a given query increases the hit ratio significantly. Thus, there is an opportunity for selecting features that can approximate an optimal admission policy. As the results of the previous section show, an admission policy using the stateful feature based on the past frequency of queries outperforms SDC. The stateless features *LenC* and *LenW* we have selected, however, were not sufficient to outperform SDC, although they still outperform LRU.

Although the stateful feature *PastF* performs well, there are two main issues with this feature. First, it requires the system to maintain the frequency of past queries, and consequently to use more memory space. Second, in very dynamic environments, the past may not correspond to the current query traffic, thus leading to poor performance. Thus, in settings with tight memory constraints or that rapidly change the distribution of incoming queries, stateless features are more appropriate. Designing stateless features that perform well proved not to be an easy task, however. We have presented only two stateless features, but we have experimented with more features which gave similar results. It is hence an open problem if there exists a feature (or combination of features) that can achieve a performance as good as the one of stateful features.



**Fig. 5.** Hit ratios of the stateless features  $LenC$  and  $LenW$  for the UK query log and an AC cache of 500K.

There are other stateless features that are potential candidates for improving the hit ratio, in particular features that benefit from data readily available in a search engine. Two such cases are the frequency of terms in the text and the frequency of the query in the text. The former needs extra memory, but that frequency information is usually already available in the index of a search engine. The latter needs extra computing time, but could be estimated quickly, in particular when there are few occurrences.

Another interesting open problem is the design of stateful features not based on frequency. It is not clear whether there are stateful features that perform at least as well as features based on frequency. For example, user sessions might contain information useful to determine queries that will be hits.

## 7 Conclusions

We have shown that a simple admission policy improves upon two well-known cache management policies: LRU and SDC. An important observation is that even a small improvement in hit ratio represents an important increase in hits for large query logs. For example, if we assume a fraction of 0.01 more hits out of 10 million queries, then we have 10 thousand more hits. Moreover, as we approach the hit ratio for an infinite cache, every small improvement is significant.

Our best admission policy uses the past frequency of queries to predict their frequency in the future. To compute the past frequencies, we considered an initial training period. Alternatively, one can use a sliding window scheme, and maintain frequency information on the current window instead of a previous training period. As for the benefits of our scheme with this feature, we obtained:

1. A relative improvement of 6% over LRU and 5% over SDC for the UK log with a cache holding results for 500k queries;
2. A relative improvement of 21% over LRU and 4% over SDC for the Altavista log with a cache holding results for 100k queries.

We have also experimented with an interesting set of policies that do not require maintaining usage information. These policies are interesting because they improved over LRU and they consume less memory resources. It is part of future work to determine if there exists such a policy based on stateless features that outperforms SDC.

There are plenty of open problems for future work. Among them we can mention:

- Combining different features to optimize the hit ratio using different machine learning techniques. This means optimizing the parameters and the weights of features;
- Using efficiently the space for different parts of the cache memory, as well as for information relevant to features;
- Modeling the behavior of such cache policies along with all other parts of a search engine to evaluate the trade-offs, in particular with respect to the infrastructure cost;
- Defining for each feature the function that establishes the relation between its threshold value and the cache size.

## References

1. Denning, P.J.: Virtual memory. *ACM Computing Surveys* **2** (1970) 153–189
2. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The Impact of Caching on Search Engines. In: *Proceedings of the 30th ACM SIGIR Conference*. (2007)
3. Markatos, E.P.: On caching search engine query results. *Computer Communications* **24** (2001) 137–143
4. Xie, Y., O’Hallaron, D.R.: Locality in search engine queries and its implications for caching. In: *INFOCOM*. (2002)
5. Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: *Proceedings of the 12th WWW Conference*. (2003) 19–28
6. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems* **24** (2006) 51–78
7. Megiddo, N., Modha, D.S.: Outperforming LRU with an adaptive replacement cache algorithm. *IEEE Computer* **37** (2004) 58–65
8. Saraiva, P.C., de Moura, E.S., Ziviani, N., Meira, W., Fonseca, R., Riberio-Neto, B.: Rank-preserving two-level caching for scalable search engines. In: *Proceedings of the 24th ACM SIGIR Conference*. (2001) 51–58
9. Long, X., Suel, T.: Three-level caching for efficient query processing in large web search engines. In: *Proceedings of the 14th WWW Conference*. (2005) 257–266
10. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: Analysis of caching and replication strategies for Web applications. *IEEE Internet Computing* **11** (2007) 60–66
11. Olston, C., Manjhi, A., Garrod, C., Ailamaki, A., Maggs, B., Mowry, T.: A scalability service for dynamic Web applications. In: *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA (2005) 56–69
12. Malik, T., Burns, R., Chaudhary, A.: Bypass Caching: Making Scientific Databases Good Network Citizens. In: *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. (2005) 94–105
13. Brehob, M., Enbody, R.: An analytical model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University (1999)