Learning to Distribute Queries into Web Search Nodes

Marcelo Mendoza, Mauricio Marín, Flavio Ferrarotti and Barbara Poblete

Yahoo! Research Latin America
Av. Blanco Encalada 2120, 4th floor, Santiago, Chile
{mendozam,mmarin,flaviof,bpoblete}@yahoo-inc.com

Abstract. Web search engines are composed of a large set of search nodes and a broker machine that feeds them with queries. A location cache keeps minimal information in the broker to register the search nodes capable of producing the top-N results for frequent queries. In this paper we show that it is possible to use the location cache as a training dataset for a standard machine learning algorithm and build a predictive model of the search nodes expected to produce the best approximated results for queries. This can be used to prevent the broker from sending queries to all search nodes under situations of sudden peaks in query traffic and, as a result, avoid search node saturation. This paper proposes a logistic regression model to quickly predict the most pertinent search nodes for a given query.

1 Introduction

Data centers supporting Web search engines are composed of large sets of processors which form clusters of computers. These systems can be seen as a collection of slave search nodes (processors) which are fed with queries by master query-receptionist/answer-provider machines called brokers. The scale of these systems – along with issues of energy consumption, hardware/software investment and maintenance costs – make it relevant to devise efficient and scalable query solution strategies. These strategies include: data structures which are suitable for distributed indexing, different levels of caching and parallel query processing algorithms. All of these factors, when operating in combination, enable search engines to cope efficiently with heavy and highly dynamic query traffic. Certainly an important design goal is to be able to reduce the level of hardware redundancy in order to allow the search engine to operate at a high utilization level in steady state query traffic, while at the same time respond in a stable manner to sudden peaks in traffic. This paper describes a method that allows search engines to achieve this last objective.

Broker machines contain a results cache which stores the answers to the most frequent user queries. We call this cache RCache. When a given query is not found in the RCache it is sent to P search nodes which respond with their top-N results for the query and then the broker merges these local top results to get the global top-N results and produce the answer set for the query. The document collection is evenly partitioned into the P search nodes and each sub-collection is indexed with the well-known inverted file which is usually kept in secondary memory. A search node can also contain additional levels of caching in its main memory. The most basic one is a cache of inverted lists, but there can also exist additional caches such as a cache of pre-computed inverted list intersections between pairs of terms frequently cooccurring in queries. The method presented in this paper assumes the existence of an additional cache – which we call LCache – in the broker machine and, possibly, another cache in each search node – which we call global top-N cache. The LCache is a *small location cache* which stores, for each cached query, the set of search node IDs that produce the global top-N results for the query.

The relationship between the RCache and LCache is as follows. As pointed out in [10], the objective of caching is, to improve the query throughput of search engines. Therefore, frequent queries which require large amounts of resources be processed, are better candidates to be hosted by the RCache, than frequent queries which require fewer resources. The goal of the LCache is to prevent these last more *inexpensive* queries from being sent to all search nodes, including also unfrequent queries that obtain their top-N from very few processors. Under heavy query traffic conditions it is critical to reduce the number of processors hit by each query. This significantly reduces the overhead and therefore increases scalability. This occurs because processor utilization is better achieved for a large number of different queries, than for fewer queries that require more processors and producing a similar work-load.

The LCache admission policy caches queries containing the largest values for the product $f \cdot L \cdot P/m$ where f is the frequency of the query, L the average cost of processing the query in a search node, and m the number of search nodes producing the documents within the global top-N results. In contrast, the queries cached in the RCache are those queries with the largest values for the product $f \cdot L \cdot m$. For the purpose of the method proposed in this paper we also keep in the LCache the search node IDs for the results of the queries stored in the RCache.

Typically a single entry in the RCache is of the order of KBs whereas a single entry in the LCache is of the order of a few bytes. Wherever the answers of the RCache are stored, in secondary memory or in another set of processors, the LCache can also store its entry contents there. The entries in the LCache can be efficiently compressed by performing document clustering and distributing co-related clusters in the same or consecutive processor IDs to balance the load properly. In addition, document IDs can be re-labeled in order to make them consecutive within each cluster so that (doc_ID, proc_ID) pairs can also be compressed efficiently. So, alternatively each entry in the LCache could be composed of a pair (doc_ID, proc_ID) which allows the query processing algorithm to go to the respective processors and directly get the snippets and other meta-data to build the result set for the query. Certainly this occupies more space than simply compressing processors IDs and thereby we prefer to keep the doc_ID values distributed onto the processors in what we call global top-N caches. The objective is to allow the broker store as much location data as possible for queries that require few processors to be completely solved.

Apart from improving overall query throughput, which has been shown in [16], the work in [9] shows that the LCache can also be used to improve the search engine ability to cope efficiently with sudden peaks in query traffic. This work resembles the idea in [19] which proposes that upon sudden peaks in traffic, the queries not found in the RCache must be sent to less than P search nodes as would have been the case when traffic is in steady state. The aim is to avoid processor saturation and temporarily respond to users with approximated answers to queries. The queries are routed to the search nodes capable of providing good approximations to the exact results.

However, this is where the similarity between [9] and [19] ends. To determine the set of processors to which the queries that are not found in the RCache must be sent, the method in [19] proposes a technique based on representation of search node contents. This method is used to dynamically rank search nodes in accordance with specific query terms. Whereas the method proposed in [9] views the query terms and search node IDs stored in the LCache as a semantic cache [4, 3]. The approach presented in [9] determines which processors are most likely to contain good approximated results for queries that are not found both in the RCache *and* in the LCache. We emphasize "and" to note that the LCache by itself contributes to reduce the work-load into processors which is critical in this case. The space used by the representation of search node contents can be used to host LCache entries with the advantage that hits in the LCache can be used to respond exact answers to queries without affecting significantly the overall work-load of the processors.

Nevertheless, ignoring the fact that the LCache is able to deliver exact answers at low processor overload cost in high query traffic situations, the LCache semantic method proposed in [9] is not as effective as the method proposed in [19]. On the average and for our experimental datasets, the method in [19] is able to rank search nodes for more than 80% of the queries, whereas in the LCache semantic method if is below 40%. In this paper we propose a new LCache-based method that matches the performance of the method proposed in [19] for this metric. Given the equivalent performance, the advantages of our method over the methods proposed in [9] and [19] are evident in practical terms since it is well-known that large Web search engines are daily faced with drastic variations in query traffic. The method proposed in this paper uses the LCache as a training dataset for a standard machine learning method. This makes it possible to build a predictive model for the search nodes capable of producing the best results for queries not found in the RCache and the LCache. The method uses a logistic regression model to predict those search nodes.

The remaining sections of the paper are organized as follows. In Section 2 we review related work. In Section 3 we introduce the learning to cache issue, modelling the location cache problem in the machine learning domain. Experimental results are shown in Section 4. Finally, we conclude in Section 5.

2 Related Work

Regarding caching strategies, one of the first ideas studied in literature was having a static cache of results (RCache abridged) which stores queries identified as frequent from an analysis of a query log file. Markatos et al. [17] showed that the performance of the static caches is generally poor mainly due to the fact that the majority of the queries put into a search engine are not frequent ones and therefore, the static cache reaches a low number of hits. In this sense, dynamic caching techniques based on replacement policies like LRU or LFU achieved a better performance. In another research, Lempel and Moran [13] calculated a score for the queries that allows for an estimation of how probable it is that a query will be made in the future, a technique called Probability Driven Caching (PDC). Lately, Fagni *et al.* [6] proposed a structure for caching where they maintain a static collection and a dynamic one, achieving good results, called Static-Dynamic Caching (SDC). In SDC, the static part stores frequent queries and the dynamic part handles replacement techniques like LRU or LFU. With this, SDC achieved a hit ratio higher than 30% in experiments conducted on a log from Altavista. Long and Suel [15] showed that upon storing pairs of frequent terms determined by the co-occurrence in the query logs, it is possible to increase the hit ratio. For those, the authors proposed putting the pairs of frequent terms at an intermediate level of caching between the broker cache and the end-server caches. Baeza-Yates *et al.* [2] have shown that caching posting lists is also possible, obtaining higher hit rates than when just doing results and/or terms. Recently, Gan and Suel [10] have studied weighted result caching in which the cost of processing queries is considered at the time of deciding the admission of a given query to the RCache.

A common factor that is found in the previous techniques is that they attempt to give an exact answer to the evaluated queries through the cache exactly. The literature shows that the use of approximation techniques like semantic caching can be very useful for distributed databases. A seminal work in this area is authored by Godfrey and Gryz [11]. They provide a framework for identifying the distinct cases that are presented to evaluate queries in semantic caches. Fundamentally, they identify cases which they call semantic overlap, for which it is possible for the new query to obtain a list of answers from the cache with a good precision. In the context of Web search engines, Chidlovskii *et al.* [4, 3] poses semantic methods for query processing. The proposed methods store clusters of co-occurring answers identified as regions, each of these associated to a signature. New frequent queries are also associated to signatures, where regions with similar signatures are able to develop their answer. Experimental results show that the performance of the semantic methods differs depending on the type of semantic overlap analyzed. Amiri *et al.* [1] deals with the problem of the scalability of the semantic caching methods for a specific semantic overlap case known as query containment.

A related work to our objective is authored by Puppin et al. [19]. The authors proposed a method where a large query log is used to form P clusters of documents and Q clusters of queries by using a co-clustering algorithm proposed in [5]. This allows defining a matrix where each entry contains a measure of how pertinent a query cluster is to a document cluster. In addition, for each query cluster a text file containing all the terms found in the queries of the cluster is maintained. Upon reception of a query q the broker computes how pertinent the query q is to a query cluster by using the BM25 similarity measure. These values are used in combination with the matrix co-clustering entries to compute a ranking of document clusters. The method is used as a collection selection strategy achieving good precision results. At the moment this method is considered as state-of-the-art in the area. This method, namely PCAP, will be evaluated against our method in the experimental results section.

The another similar work to our objective is our prior work [9]. Here we propose proposed using an LCache as a semantic cache. The proposed method allows reducing the visited processors for queries not found in an LCache. The semantic method for evaluating new queries uses the inverse frequency of the terms in the queries stored in the cache (Idf) to determine when the results recovered from the cache are a good approximation to the exact answer set. The precision of the results differs depending on the semantic overlap case analyzed. In the context of this paper we call this method SEMCACHE and it will be evaluated against our method in the experimental results section. Recently [16], we show that the combination of an LCache and an RCache using dynamic caching strategies is able to achieve efficient performance under severe peaks in query traffic, as opposite to the strategy that only uses the RCache that gets saturated as they are not able to follow the variations in traffic.

To the best of our knowledge, there is no previous work that uses machine learning methods to predict the most likely search nodes for queries that are not cached in the broker machine.

3 Learning to Rank Search Nodes

3.1 Modelling the Location Cache Problem in the Machine Learning Domain

A classification task is based on the appropriate processing of training and evaluation datasets. A training dataset consists of data instances, each of which are composed of a collection of features and one or more tags that indicates which category the instance belongs to. A training dataset is used to construct a classification model that allows it to categorize new data instances. An evaluation dataset is used to measure the classifier's performance. Each data instance from the evaluation dataset is categorized using the classifier. Each of these data instances contains one or more labels that indicate which category they belong to. They are compared to the categories predicted by the classifier.

The LCache's problem could be modeled in the machine learning domain like a classification task. Let LC_i be an entry in an LCache. An entry LC_i is formed by the query terms q_i stored in that position, and by the list of machines that allows the top-N results to be obtained for q_i. Given a new query q (not stored in the cache) we want to determine the list of machines that allows the top-N results of q to be obtained. In the machine learning domain we can interpret each machine as a category. We can also interpret each LC_i entry from the compact cache as an instance of training data. From these, we prepare a collection of features (the terms of q_i) and the categories to which it (the list of machines) belongs. Given that the distributed search engine typically considers multiple machines, the problem in the machine learning domain is multi-class. In addition, given that the responses to the query could be distributed into several machines, we need the predictive model to give us a list of likely machines. Due to this factor, the problem is also multi-labeled.

Given an LC_i entry in the LCache, we will call the list of terms of q_i $\mathbf{x}_i$ and the list of machines $\mathbf{y}_i$. Given an LCache of l entries, we arrange a training dataset formed by pairs $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1, \ldots, l$. We also consider that the LCache vocabulary (the group of terms made from the queries stored in the LCache) contains n elements, and that the distributed document system considers m machines. Later, $\mathbf{x}_i \in \{0,1\}^n$, where 1 indicates that the term was used to form the query, 0 in another case, and $\mathbf{y}_i \in \{0,1\}^m$, where 1 indicates that the machine contains one of the top-N responses for q, 0 in another case.

The predictive model generally consists of a $y(\mathbf{x}, \mathbf{w})$ function where $\mathbf{w}$ corresponds to the parameters of a combination (generally convex) over the vector of features $\mathbf{x}$ of the entry. In the simplest case, the model is linear to the parameters $\mathbf{w}$ and so $y = \mathbf{w}^T\mathbf{x} + w_0$. The coefficient w_0 corresponds to the model's bias. To decide each category, an activation function is applied to y, which we will call $f(\cdot)$. Frequently, the activation function corresponds to the sign function which makes it possible to evaluate the categorization based on a Boolean variable.

3.2 Logistic Regression Model

For the first approximation we will consider the classification problem restricted to two classes. The construction of a classification model consists of the estimation of the following conditional probability:

$$\mathbf{P}(y = \pm 1 \mid \mathbf{x}, \mathbf{w}).$$

Notice that the probability is conditioned to $\mathbf{x}$, meaning the feature vector of the entry instance that we want to classify. It is also conditioned for the predictive model represented by the parameter vector $\mathbf{w}$.

The logistic regression model assumes that the conditional probability can be calculated in the following way:

$$\mathbf{P}(y = \pm 1 \mid \mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-y(\mathbf{W}^T\mathbf{X} + w_0)}},$$

that corresponds to an activation function on the predictive model known as a logistic function with trajectory on $[0,1]$. The classification model could be estimated minimizing the negative log-likelihood function (loss function) as follows:

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\log(1 + e^{-y_i(\mathbf{w}^T x_i + w_0)}),$$

where $C > 0$ corresponds to the penalization parameter of the classification errors and the pairs $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1, \ldots, l$ correspond to the training dataset. The factor $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ is known as the regularization factor. It allows the classification model to have good generalization properties.

The logistic regression model can be successfully applied to many text categorization problems due to the fact that it is scalable on high dimensional data. In the compact cache problem domain we will consider the probability $\mathbf{P}(y_m = \pm 1 \mid \mathbf{x}_q, \mathbf{w})$ where $\mathbf{x}_q$ represents the features of the new query q and y_m represents the event *to process the query q in the m machine.* In the following section we will show how we can use the logistic regression model for the multi-class multi-label case.

3.3 Multi-class Multi-label Logistic Regression

One form of extending the logistic regression model for multi-class problems is to use the method one-vs.-rest (OVR). This method consists of developing a binary classifier for each class that allows the objective class to be separated from the rest of the classes. Later we provide a probability $\mathbf{P}(y_m = \pm 1 \mid \mathbf{x}_q, \mathbf{w})$ for each category.

In the compact cache domain problem using OVR approximation for each new query, the predictive model gives a list of probabilities, one for each machine. This is done by modelling the new query in the feature space defined by the LCache (the terms that constitute the vocabulary for all the queries stored in the LCache) and calculating the logistic activation function for each machine in the distributed system. For each machine, we have a predictive model that allows us to indicate how promissory the machine is for the new query. The classification model then determines a list of probabilities associated with each machine, which indicates the order in which the machines must be visited in order to retrieve relevant results.

One of the strong points of the OVR method is the fact that the multi-class methods existing in literature today are generally more costly and complex to implement, leading to a slow training phase. The OVR methods have shown comparable precision to the multi-class methods with faster training times.

Since a binary classification method will be used but the training data is multi-label, we must utilize a data transformation method. Following Tsoumakas *et al.* [20] each instance of training data $(\mathbf{x}_i, \mathbf{y})$, will be decomposed of z uni-label instances $(\mathbf{x}_i, \mathbf{y}_m)$, $m = 1, \ldots, z$, where z corresponds to the number of categories that the query q_i belongs to (the length of the list of machines that allows the top-N results for q_i to be obtained).

4 Experimental Results

4.1 Dataset and Document Distribution

The dataset we will use in this paper corresponds to a query log file of Yahoo! from the year 2009. The query log contains 2,109,198 distinct queries and 3,991,719 query instances. 1,629,113 distinct queries were considered as training data, leaving the 602,862 remaining queries for evaluation. The testing set considers 999,573 query instances in the query log trace. The training set considers 2,992,146 query instances. The vocabulary of the 3.9 million query instances is compound by 239,274 distinct query terms.

We initialize the LCache with the 150,000 most frequent queries in the training query log trace. The LCache contains 204,485 different terms. Out of the 602,862 distinct queries in the testing dataset, there are 453,354 queries that have at least one term in the set of terms of the LCache.

For each query considered in the query log data, we obtained the top-50 results retrieved using the Yahoo! Search BOSS API [21]. The generated document collection corresponds to 50,864,625 documents. The API uses the Yahoo! services to get the same answers that are presented to actual users by the production Yahoo! search engine.

The document collection was distributed using a query-document co-clustering algorithm. The co-clustering algorithm allows us to find clusters of related documents. We choose to use this clustering algorithm because we wanted to compare our results with those of PCAP - the strategy of collection selection that is considered as state-of-the-art in the area [19]. A **C++** implementation of the algorithm was used for the experiments. The source code of this algorithm is available in [18].

The document collection was distributed over two arrays of processors considering 16 and 128 machines. Each entry in the LCache stores the set of processors that have the top-20 results for the corresponding query.

4.2 Performance Evaluation

Performance measures To evaluate the performance of our method we compare the results coming from the predictive model with the results coming from the Yahoo! Search Engine. This is an effective approach to evaluate the performance in our distributed search engine because the document collection was built using the Yahoo! Search BOSS API. Thus, the best case for our method is to predict the machines where the top-N results for the testing queries are.

Following Puppin *et al.* [19], we will measure the effectiveness of our approach using the *Intersection* measure defined as follows:

Intersection : Let's call G_q^N the top-N results returned for a query q by the Yahoo! Search Engine, and H_q^N the top-N results returned for q by our method. The intersection at N, $INTER_N(q)$, is the fraction of results retrieved by the the proposed method that appear among the top-N documents in the Yahoo! Search Engine BOSS API:

$$INTER_N(q) = \frac{|H_q^N \cap G_q^N|}{|G_q^N|}.$$

The $INTER_N$ measure for a testing query set is obtained as the average of the $INTER_N(q)$ measured for each testing query.

Implementation of the methods For the evaluation we will consider the state-of-the-art approximated methods PCAP and SEMCACHE, described in Sections 2 and 3, respectively. We tested the correctness of our PCAP implementation by using the query training set over the array of 16 processors. In this setting, PCAP obtained a $INTER_{20}$ measure of 31.40%, 43.25%, 59.19%, and 77.68%, when we visit 1, 2, 4, and 8 processors, respectively. These $INTER_{20}$ values are very similar to those obtained by Puppin *et al.* [19] in their experiments. Regarding the SEMCACHE method, we use an Idf threshold equals to 10, as it is recommended in [9].

Our method (LOGRES abridged) was implemented using a trust region Newton method that has been proved on high dimensional data [14]. The method is based on a sequential dual method suitable for large scale multi-class problems [12] that reduces the time involved in the training process. The logistic regression implementation used in these experiments is available on [8].

For the training phase of LOGRES, we perform a 5-fold cross validation process to determine the value of the C parameter. We choose the value of C measuring the accuracy of LOGRES over an array of 16 and 128 processors. Results are shown in Table 1.

Penalty factor	16 Machines	128 Machines
C = 0.00001	15.2067	7.9573
C = 0.0001	15.2067	7.9573
C = 0.001	15.2067	7.9573
C = 0.01	**15.2067**	**7.9573**
C = 0.1	14.9742	7.9411
C = 1	10.0769	6.5423
C = 10	9.7191	5.2433

Table 1. LOGRES 5-folds cross-validation training accuracy for differents penalty factors (percentages). Bold fonts indicate best performance values.

Results We use the 602,862 distinct queries reserved for testing to evaluate the *INTER* measure. As it is shown in Table 1, SEMCACHE approximates a fraction of these queries (the 39.75% of the testing set match a semantic overlap case). On the other hand, PCAP approximates the 87.56% of the testing set (the remaining testing queries do not match terms with the query-document clusters). LOGRES approximates to the 75.27% of the testing set (we need at least one term of the query shared with the LCache feature space to build the query vector representation). Notice that for the SEMCACHE and LOGRES

	Number of Queries	Percentage
PCAP	527,918	87.56
SEMCACHE	239,682	39.75
LOGRES	453,354	75.27

Table 2. Testing set coverage for the methods.

methods we are not considering hits in the LCache (app 4.6% of the testing dataset) because we are evaluating only the predictive capacity of each method.

As Table 1 shows, the coverage over the testing set differs depending on the method evaluated. To perform a fair comparison between these methods, we calculate the *INTER* measure over the whole query testing dataset, sending the queries not covered by the methods to all processors in a random order.

For the 16 processors array, we calculate the *INTER* measure for 1, 2, 4 and 8 processors, at N = 5, 10, and 20. For the 128 processors array, we calculate the measure for 8, 16, 32 and 64 processors, at N = 5, 10, and 20. As a baseline we consider a random order generator (RANDOM abridged). Table 2 shows the results obtained for 16 and 128 processors.

$INTER_5\%$

	1	2	4	8	16
RANDOM	6.30	12.55	25.08	49.98	100
PCAP	22.38	33.12	48.31	69.61	100
SEMCACHE	10.26	17.49	30.54	54.50	100
LOGRES	13.29	20.16	32.46	55.09	100

$INTER_5\%$

	8	16	32	64	128
RANDOM	6.21	12.51	24.81	49.81	100
PCAP	16.48	23.58	36.64	64.03	100
SEMCACHE	11.68	19.17	32.55	57.66	100
LOGRES	16.81	24.46	37.72	65.88	100

$INTER_{10}\%$

	1	2	4	8	16
RANDOM	6.29	12.55	25.08	49.99	100
PCAP	22.85	33.82	49.05	70.01	100
SEMCACHE	10.27	17.46	30.48	54.43	100
LOGRES	12.72	19.31	31.32	54.06	100

$INTER_{10}\%$

	8	16	32	64	128
RANDOM	6.20	12.51	24.85	49.88	100
PCAP	14.56	21.36	34.34	62.31	100
SEMCACHE	11.24	18.65	31.97	57.24	100
LOGRES	15.11	22.27	34.82	63.90	100

$INTER_{20}\%$

	1	2	4	8	16
RANDOM	6.28	12.53	25.05	49.99	100
PCAP	23.77	35.11	50.51	71.01	100
SEMCACHE	10.44	17.66	30.67	54.59	100
LOGRES	12.21	18.58	30.29	53.10	100

$INTER_{20}\%$

	8	16	32	64	128
RANDOM	6.19	12.48	24.84	49.89	100
PCAP	13.21	19.83	32.81	61.06	100
SEMCACHE	11.01	18.37	31.65	56.97	100
LOGRES	13.56	20.16	31.94	61.90	100

Table 3. Performance evaluation of the methods over a distributed search engine of 16 and 128 processors

PCAP outperforms LOGRES in the 16 processors array by approximately 10% when we visit the first and second machine. This difference increases when we visit 4 and 8 machines. When we tested the methods over an array of 128 processors, we can see that PCAP degrades by a considerable margin. PCAP degrades its performance by approximately 6% when we visit the first 8 processors. This difference increases when we visit more machines. On the other hand, LOGRES improves its performance by approximately 3% when we visit the first 8 and 16 processors. This improvement increases when we visit more processors. These results show that LOGRES scales well with regards to the number of processors.

Notice that in this comparison we are using a very small LCache and we are not considering hits in the LCache. The comparison was performed by only considering the predictive capacity of each method, but in the cases of SEMCACHE and LOGRES, this means that we discard 27,934 testing queries that made hit in the LCache (42,833 query instances in the query log trace).

5 Conclusions

We have presented the compact cache problem by dealing with it from the machine learning perspective. We have shown that each entry in an LCache can be used as a training instance for a standard machine learning technique in which we use the query terms as features and each machine is represented by a category. The problem corresponds to a multi-class problem (m machines are considered, $m > 2$, in which the documents in the collection are distributed) and multi-label (given a query q, the top-N responses for q can be distributed in several machines).

As more entries are added to the compact cache, the size of its vocabulary, (i.e., the number n of different terms needed to formulate all queries stored in the cache) also grows, which in turn, increases the dimensionality of the feature space of the corresponding learning machine. On the contrary, for the problem studied in this work, the number of machines m among which the document collection is partitioned, keeps constant. Furthermore, the number of different terms stored in the cache is usually of the order of 10^5 while the number of machines is of much smaller order, usually 10^2. The literature shows that for this problem setting linear classifiers offer good results. This is due to the high dimensionality of the data allowing linear separability which is in general the case of the text. In this scenario, the state of the art shows that there are several linear classifiers that could be useful for learning cache. Among these we find the linear Support Vector Machine (linear SVMs) and the recently presented Logistic Regression. We decide to target the problem by using logistic regression, because logistic activation function allows a probability to be estimated for each machine. This favors the definition of an order in which visiting the machines is recommended, unlike the rest of the linear classification models which frequently use the sign function for the activation.

An advantage of the LOGRES method in regard to the other evaluated methods is the small amount of space that it occupies. Strictly speaking, LOGRES could even do without the LCache, storing only the parameters of the predictive model for each machine like we showed in the experiment section. This obtained a predictive capacity comparable to the capacity of PCAP. When the LCache is added to the broker, the performance of the method would be even better because a large proportion of queries would hit the LCache. Given that the vector representation of each query is sparse, the method involves low computational costs.

References

1. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *ICDE*, 2003.
2. R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. ACM TWEB, 2(4), 2008.
3. B. Chidlovskii, C. Roncancio, and M. Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. Computer Networks, 31(11-16):1347-1360, 1999.
4. B. Chidlovskii, and U. Borghoff. Semantic Caching of Web Queries. VLDB Journal, 9(1):2-17, 2000.
5. I. Dhillon, S. Mallela and D. Modha. Information-theoretic co-clustering. In *KDD*. 2003.
6. T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. ACM TOIS, 24(1):51-78, 2006.
7. F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti. A Metric Cache for Similarity Search. In *LSDS-IR*, 2008.
8. R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research, 9:1871-1874, 2008
9. F. Ferrarotti, M. Marin and M. Mendoza. A Last-Resort Semantic Cache for Web Queries. In *SPIRE*, 2009.
10. Q. Gan, T. Suel, Improved Techniques for Result Caching in Web Search Engines. In *WWW*, 2009.
11. P. Godfrey, and J. Gryz. Answering Queries by Semantic Caches. In *DEXA*, 1999.
12. S. Keerthi, S. Sundararajan, K. Chang, C. Hsieh, and C. Lin. A sequential dual method for large scale multi-class linear SVMs. In *SIGKDD*, 2008.
13. R. Lempel, and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, 2003.
14. C. Lin, R. Weng, and S. Keerthi. Trust region Newton method for large-scale logistic regression. Journal of Machine Learning Research, 9:627-650, 2008.
15. X. Long, and T. Suel. Three-level caching for efficient query processing in large Web search engines. In *WWW*, 2005.
16. M. Marin, F. Ferrarotti, M. Mendoza, C. Gomez, and V. Gil-Costa Location Cache for Web Queries. In *CIKM*, 2009.
17. E. Markatos. On caching search engine query results. Computer Communications, 24(7):137-143, 2000.
18. D. Puppin, and F. Silvestri. C++ implementation of the co-cluster algorithm by Dhillon, Mallela, and Modha. Available on *http://hpc.isti.cnr.it*
19. D. Puppin, F. Silvestri, R. Perego, and R. Baeza-Yates. Load-balancing and caching for collection selection architectures. In *INFOSCALE*, 2007.
20. G. Tsoumakas, I. Katakis. Multi-label Classification: An Overview. International Journal of Data Warehousing and Mining, 3(3):1-13, 2007.
21. Yahoo! Search BOSS API. Available on *http://developer.yahoo.com/search/boss/* , 2009.
22. H. Yan, S. Ding and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.