Abstract

“History Teaches Everything, Including the Future”, wrote Alphonse de Lamartine in the nineteen century. Even if history cannot be really considered a predictive science, historical information can successfully be used in many fields. This paper deals with Web Search Engines and their Query logs, which contain historical information about past usage of such systems. We will present some of the most interesting results obtained in this field by the High Performance Computing Lab in Pisa in collaboration with Research Labs worldwide. The techniques reviewed are mainly focused on enhancing the efficiency of large-scale distributed search systems.

1 Introduction

Web Search Engines store requests users submit in Query Logs. In some sense Query Logs represent the history of the search system from the perspective of the engine and are a very precious resource. Indeed, they can be considered as one of the most important source of knowledge that is routinely exploited to improve performance in terms of both effectiveness and efficiency of a search engine.

The title of this paper paraphrases a quote from Alphonse de Lamartine that in the past has picked by many historian to justify the study of history from a scientific perspective. That is, they were arguing that a scientific formalization of history would have allowed people to foresee future events according to what happened in the past.

Query logs, as a source of historical information, do help in predicting future usage patterns in Search Engines. As a matter of fact, there are many important information that can be devised from aggregate statistics drawn out of query log data. The ranking subsystem, for instance, uses click-through information to score results (i.e. the more a page is clicked the more important it is). History on past queries can be used to depict user profiles. These profiles are, in turn, used to classify new users and to supply a more personalized service to them (e.g. suggesting queries of potential interest, re-ranking pages according to their personal preferences).

Under the hood, search systems use historical information also to speed-up those operations that have to be frequently carried out. Power-laws regulate query streams. Therefore, accelerating the most frequent operations correspond, basically, to speed-up a large fraction of the total operations performed [14].

In this paper we are going to review some of the most recent state-of-the-art techniques developed by us at the High Performance Computing Lab in Pisa in collaboration with Research Labs worldwide.

Section 2 deals with the analysis of past “Query Sessions” with the aim of storing the results of frequently requested queries into a result cache for speeding-up the retrieval of the same query in the future. Index partitioning can benefit from the knowledge extracted from past queries: Sections 3, and 4 deal with this important aspect in distributed Search Engines. A new kind of
Search Service is starting to appear in the digital ecosystem: similarity-based multimedia search engines. Section 5 deals with caching in similarity-based search applications.

2 Caching

Caching is the main mean with which systems exploit memory hierarchies. There’s a whole body of literature on systems where caching is extensively studied.

Caching in web search engine [7] is, basically, a matter of stocking either results, partial results, or raw posting lists, into a smaller, and faster to lookup, buffer memory. Usually, in real systems, caching uses a combination of the three above kinds, and the final system appears as in Figure 1.

The right to decide what results are to be kept in cache is acknowledged to the caching policy. In case of the cache running out of space, the caching policy is responsible for expunging a result in favor of another retained to be more likely to be requested in future.

![Figure 1: Caching module placement within the typical structure of a Web Search Engine.](image)

In Web search engines, caching has been studied since 2000 when Markatos presented the (probably) first work specifically targeted on exploiting caching possibilities in Web search engines [9].

The setting with respect to query search results caches in Web search engines consists of result pages of search queries to be cached. A search query is defined as a triplet \( q = (qs, from, n) \) where \( qs \) is a query string made up of query terms, \( from \) denotes the relevance rank of the first result requested, and \( n \) denotes the number of requested results. The result page corresponding to \( q \) would contain the results whose relevance rank with respect to \( qs \) are \( from, from + 1, \ldots, from + n - 1 \). The value of \( n \) is typically 10.

### Caching of Query Results

Caching in Web search engines immediately recalls the storing of results of previously computed queries in a faster memory area. This kind of caching is known as: caching of search results. A good deal of research has been (and it is currently) conducted on how efficiently manage the (small) space dedicated to the cache in an optimal way.

Roughly speaking, caching should exploit regularities in query streams to keep copies of “likely-to-be-accessed-in-the-future” queries.

The caching design space is two-dimensional: choosing the caching policy in order to increase as much as possible the hit-ratio, and optimizing the architecture of the caching system to improve response time and throughput (number of queries per second) of the search engine. Both dimensions are important since higher hit-ratios often correspond to lower response times and higher throughput.

In the SDC policy, proposed by Fagni et al. in [5], it has been shown how to effectively exploit past queries in caching. SDC is the acronym of Static Dynamic Caching since it basically integrates two types of caching: static, and dynamic. Static caching was, indeed, previously analyzed by Markatos in [9] where it was shown that a static-only cache of query results badly performs on the tested log. The merit of SDC is to have mixed the two concept of Static and Dynamic caching trying to balance the benefits in terms of capturing frequent queries by means of the Static caching, and recent queries, by means of the Dynamic policy. In the static cache, the set of the most-frequently-submitted-in-the-past queries is kept. The dynamic cache is a cache managed through a traditional replacement policy. SDC management complexity is constant, i.e. \( O(1) \), whenever a constant policy is used in the dynamic section. Furthermore, SDC introduces Adaptive Prefetching that exploits a particular and peculiar characteristics of Web search users behavior: when a user go through the \( i \)-th \((i \geq 2)\) page of results then he will, with high probability, explore also the \( i+1 \)-th page. Adaptive prefetching consists of performing prefetching whenever a request refers to the \( i \)-th \((i \geq 2)\) result pages. That is we fetch more that one result page whenever we receive a request for the second, or greater, page of results.

SDC resulted to be superior on hit-ratios achieved by a search system with a traditional completely dynamic policy. For example, on a cache table containing 256,000 results from the Altavista log, the hit-ratio of a pure LRU is about 32%, while SDC with LRU as the dynamic policy on the 40% of the cache size (i.e. about 100,000 results) obtains a hit-ratio of about 55%
when a prefetching factor of 10 is used. In [5] the authors claim the superiority of SDC being due to the way it exploits the power law in the query log. Queries that frequently appears, not necessarily present the recency property (i.e. the same frequent query might be submitted after an arbitrary large number of distinct queries). If this assumption is true, and if the LRU queue is not large enough, then some frequent query could be evicted before they will be requested again\(^1\). To assess this, Figures 2a, and 2b report the cumulative number of occurrences of each distance, measured as the number of distinct queries received by AltaVista and Yahoo! in the interval between two successive submissions of each frequent query [5, 1]. From the two figures in 2 we can conclude that even if we set the size of a LRU cache to a relatively large number of entries, the miss rate will be high anyway.

\[\text{Figure 2: Cumulative number of occurrences of each distance, measured as the number of distinct queries received by AltaVista and Yahoo! in the interval between two successive submissions of each frequent query.}\]

Policies exploiting historical usage information might suffer of data model staling. The model built over a period might be not valid anymore in the next period. In [1] a static cache containing the 128,000 most frequent queries from a Yahoo! log has been used to test the hit-ratio trends on an hourly basis. Figure 3 shows that the hit-ratio is quit stable (ranging from 0.25 to 0.35) within a period of at least a week (for this particular query log, obviously).

As said above, caching of query results is, most likely, the first thing coming to your mind when speaking about caching in Web search engines. Posting list caching also exists, and it is as important as result caching.

Posting lists are used during the computation of query results. In less recent search systems, posting lists were thought to be stored on disk. In more modern Web search systems, anyway, it is very likely that the entire posting lists would be stored on memory. In both cases, anyway, having a more compact and faster structure storing frequently accessed lists might be of help for improving query answering time. This is the idea behind posting list caching: storing frequently accessed lists to reduce delays due to list seek and retrieval operations.

Furthermore, posting list caching and query result caching are not exclusive: if both used “cum grano salis”, overall performance might end up being sensitively improved [1]. Due to space limitations we refer the reader to [1] for further information.

Caching is not just a matter of improving hit-ratio of its policy. Indeed, optimal policies for graph based workload models are possible at the cost of a linear time in the number of cache entries. A high management cost may jeopardize the benefit of a high hit-ratio. Furthermore, in a real setting on the same machine many different index server instances run in parallel (via multi-threading) and it is inconceivable to think of several private copies of the cache for every index server. While a shared cache reduces the space occupancy, it introduces the need of a regulated concurrent access to the cache structure by each thread, i.e. a spin-lock around the cache. Therefore, each access implies: lock acquisition, cache management (in case of miss the management time is higher due to the querying phase), lock release. The more efficient the cache management phase, the higher the scalability of the querying system as more concurrent tasks can run in parallel.

Having a static caching policy is important since, being the buffer read-only, it does not require any management operation and therefore, no lock is needed around the shared cache. This impacts heavily on the cache performance. In [5] a test assesses the throughput of a SDC cache under two different conditions: a lock around all the cache, a lock only on the dynamic set. Note that this last setting is the one which realistically should be used to manage concurrent accesses to a SDC cache.

Figure 4 reports the results of some of the tests con-
Figure 4: Throughput of a caching system (in number of queries served per second) for $f_{\text{static}} = 0.6$ as a function of the number of concurrent threads and different locking policies [5].

Conducted. In particular, the figure plots, for $f_{\text{static}} = 0.6$ and no prefetching, the throughput of an SDC caching system (i.e., the number of queries answered per second) as a function of the number of concurrent threads contemporaneous and concurrently accessing the cache. The two curves show the throughput of a system when each thread accesses in a critical section either the whole cache (dashed line) or just the Dynamic Set (solid line). Note that locking the whole cache is exactly the mandatory behavior of threads when accessing a purely dynamic cache (i.e., $f_{\text{static}} = 0$). Throughput was measured by considering that a large bunch of 500,000 queries (from the Tiscali log) arrives in a burst. The size of the cache was 50,000 blocks (quite small, though), while the replacement policy considered was SLRU.

The presence of the Static Set permits to approximately double the number of queries served per second. Moreover, the caching system does not only provides high throughput but can also sustain a large number of concurrent queries. Performance starts degrading only when more than 200 queries are served concurrently.

Obviously the same argument holds also in case of posting list caching.

3 Document Prioritization

Document partitioning is the most commonly used strategy for parallelizing the search process in large-scale WSEs. In a document-partitioned organization, the broker may choose among two possible strategies for scheduling a query. A naïve, yet very common, way of scheduling queries is to broadcast each of them to all the underlying IR cores. This method has the advantage of enabling an almost perfect load balancing among all the servers. On the other hand it has the major drawback of using all the servers for each query submitted. The other possible way of scheduling is to choose, for each query, the most authoritative server(s), thus reducing the number of IR cores queried. Relevance of each server to a given query has to be computed by means of a collection selection function that is built, usually, upon statistics computed over each sub-collection.

We investigated in depth the possibility of exploiting the knowledge on the past usage of the system to drive the assignment of documents, with the goal of putting together the documents that answer a given query [13]. We do this, first, by representing each document as a query-vector, i.e. a (sparse) vector listing the queries to which it answers, weighted with the search score and, second, by performing co-clustering on the query-document contingency matrix. The resulting document clusters are used to partition the documents among the servers, while the query clusters are used to guide our collection selection strategy.

Co-clustering. Typically, partitioning and selection strategies are based on information gathered from the document collection. Partitioning, in particular, is either random, or based on clustering documents, e.g., using k-means. In both cases, documents are partitioned without any knowledge of what queries will be like. We believe that information and statistics about queries may help in driving the partitions to an optimal choice. Our goal is to cluster the most relevant documents for each query in the same partition.

The cluster hypothesis states that closely associated documents tend to be relevant to the same requests. Clustering algorithms, like the k-means method cited above, exploit this claim by grouping documents on the basis of their content. We instead based our method on co-clustering queries with the documents returned in reply to each one. The algorithm we adopt is described in [4] and is based on a model exploiting the joint probability of picking up a given couple $(q,d)$, where $q$ is a given query and $d$ is a given document. All these probabilities values are collected into a contingency matrix.

Given a contingency matrix, co-clustering is the general problem of performing simultaneously a clustering of columns and rows, in order to maximize some clustering metrics (e.g. inter-cluster distance).

The way we consider documents for co-clustering can be seen as a new way of modeling documents. So far, two popular ways of modeling documents have been proposed: bag-of-words, and vector space. Since we know which documents are given as answers to each query, we can represent a document as a query-vector.
Query-vector model. Let \( \Phi \) be a query log containing queries \( q_1, q_2, \ldots, q_m \). Let \( d_{i1}, d_{i2}, \ldots, d_{in} \) be the list of documents returned as results to query \( q_i \). Furthermore, let \( r_{ij} \) be the rank value associated with the pair \((q_i, d_j)\). A document \( d_j \) is represented as an \( m \)-dimensional vector \( \delta_j = [\chi_{ij}]^{T} \), where \( \chi_{ij} \in [0,1] \) is the rank of documents \( d_j \) returned as an answer to query \( q_i \). The entries of \( \chi_{ij} \) are then normalized in order to sum to 1.

According to this query-vector models, documents that are not hit by any query in the query log are represented by null query-vectors. This is a very important feature of our model because it allows us to remove more than half of the documents from the collection without losing precision. This will be described in detail below.

The contingency matrix introduced above, can now be formally defined as \( \Upsilon = [\delta_{ij}]_{1 \leq i \leq n} \) where \( n \) is the number of distinct documents that are returned as answers to the queries.\(^2\) For each \( i, j \) each entry \( \Upsilon_{ij} = r_{ij}/ \sum_{i \in I} \sum_{j \in \Phi} r_{ij} \) is the rank of the document \( d_j \) for the query \( q_i \) normalized so that \( \Upsilon \) entries sum up to one. The contingency matrix just defined can be used into the co-clustering algorithm to obtain the document clusters identifying the partitions.

Furthermore, co-clustering considers both documents and queries. We thus have two different kind of results: (i) groups made of documents answering to similar queries, and (ii) groups of queries with similar results. The first kind of results is used to build the document partitioning strategy, while the second is the key to our collection selection strategy (see below).

The result of co-clustering is a matrix \( \hat{P} \) defined as:

\[
\hat{P}(qc_a, dc_b) = \sum_{i \in qc_a} \sum_{j \in dc_b} r_{ij}
\]

In other words, each entry \( \hat{P}(qc_a, dc_b) \) sums the contributions of \( r_{ij} \) for the queries in the query cluster \( a \) and the documents in document cluster \( b \). We call this matrix simply PCAP. The values of PCAP are important because they measure the relevance of a document cluster to a given query cluster. This induces naturally a simple but effective collection selection algorithm.

We used the ideas illustrated above to design a novel document-partitioned parallel and distributed WSEs. Our strategy is as follows.

First, we train the system with the query log of the training period, by using a reference centralized index to answer all queries submitted to the system. We record the top-ranking results for each query. Then, we perform co-clustering on the query-document matrix. The documents are then partitioned among several servers according to the results of clustering. Note that, besides the clusters identified by our strategy, we also need a further cluster containing the documents that are not returned by any query.

For querying our document-partitioned PIRS, we perform collection selection. The servers holding the selected collections are queried, and results are merged. In order to have comparable document ranking within each core, we distribute the global collection statistics to each server. So, the ranking functions are consistent, and results can be very easily merged, simply by sorting documents by their rank.

We keep logging the results of each query, also after the end of the training period, in order to further train the system and also to accommodate any topic shift. Topic shift refers to the fact that, over time, the interests of the users of a search engine can change. For instance, in the case of an unexpected calamity, there can be a sudden increase of queries about the issue.

To adjust to this possibility, the co-clustering algorithm can be periodically performed by an off-line server and documents can be moved from one cluster to another in order to improve precision and reduce server load. One interesting thing is that there is no need for a central server running a centralized version of the search engine because the rank returned by the individual server is consistent with the one a central server would return.

Collection Selection. Our collection selection strategy is based on the PCAP matrix returned by the co-clustering algorithm. The queries belonging to each query cluster are joined together into query dictionary files. Each dictionary file stores the text of each query belonging to a cluster, as a single text file. When a new query \( q \) is submitted, we use the TF.IDF metric to find which clusters are the best matches: each dictionary file is considered as a document, which is indexed with the usual TF.IDF technique. This way, each query cluster \( q_{ci} \) receives a score relative to the query \( q \) \( (r_q(q_{ci})) \). Note that even if we used TF.IDF, we could employ any other ranking metric computed on the basis of text statistics.

This ranking is finally used to weight the contribution of PCAP \( \hat{P}(i,j) \) for the document cluster \( dc_j \), as follows:

\[
 r_q(dc_j) = \sum_i r_q(q_{ci}) \times \hat{P}(i, j)
\]
In this way, we determine which core holds the most relevant documents for each query. Such knowledge is used by our load-driven routing [13], which consists in assigning a priority to each query for every core. The computing cores will answer on the basis of queries’ priorities and of their instant load, this way reducing the computing pressure for low priority queries on overloaded servers.

We performed tests using the WBR99 collection. WBR99 consists of 5,939,061 documents, about 22 GB uncompressed, representing a snapshot of the Brazilian Web (domains .br) as spidered by www.todobr.com.br. It comprises about 2,700,000 different terms. We could also use the query log of www.todobr.com.br for the period January through October 2003.

Due to the nature of data, we do not have a list of human-chosen relevant documents for each query. WBR99 includes only 50 evaluated queries. Thus, we consider the top-ranking pages returned by a central index to be relevant. In particular, when measuring precision at 5, we consider only the top five documents to be relevant. Similarly, for precision at 10 and so on.

For our experiment, we used Zettair\(^3\), a compact and fast text search engine designed and written by the Search Engine Group at RMIT University. We modified it so to implement our collection selection strategies (CORI and PCAP).

### Experimental Results

Due to limited space, we cannot show that our partitioning strategy outperforms the random allocation when using CORI. Also, we have evidence supporting the fact that our partitioning greatly outperforms a partitioning based on k-means. Here, we measure the performance of our collection selection strategy w.r.t. CORI. In this case, we test two different allocation strategies on the same document allocation, as generated by the co-clustering algorithm. For precision at 5, we consider only the five top-ranking documents (on the full index) to be relevant. Similarly, for precision at 10 we observe the top 10, and so on.

The first experimental results (Table 1) show that, in the fourth week (the first after training), PCAP is performing better than CORI: the precision reached with the first cluster is improved of a factor between 11% and 15% (highlighted entries).

We also proved that the training is robust to topic shift. We did this by using the fifth week (the second after training) as our benchmark. We did not measure great differences in precision.

### 4 Smart Term Partitioning

Term partitioning is the symmetrical approach to document partitioning. According to this partitioning strategy, the set of terms occurring in the index, i.e. the lexicon, is partitioned among the servers, and each server is able to discover only the documents containing a subset of the lexicon. Thus, for each query the broker has to select the servers holding the inverted lists relative to each of the terms of the query, and it must receive the complete results lists provided by each server in order to compute properly the the most relevant documents. However, the strategies proposed so far suffer from a significant load imbalance, due to the skewed distribution of terms in user queries and indexed documents.

Due to recent proposals of a pipelined architecture, the term partitioning approach is now attracting some attention again [11, 10]. Moffat et al. [11] introduced a novel pipelined query evaluation methodology, based on a term-partitioned index, in which partially evaluated sub-queries are passed through the servers that host the query terms. While still suffering poor load balance, this approach seems to have a great potential. On a large set of queries, pipelined query evaluation outperformed by more than 20% the document-partitioned counterpart.

The load balancing problem was also addressed later in [10], where the authors exploited both term frequency information and postings list replication to improve load balancing in their pipelined WSE.

Our proposal goes exactly in the same direction. First, we introduced a general performance model of a term partitioned WSE, where, differently from previous works, we require a good partitioning of the lexicon not only to maximize the throughput by evenly balancing

<table>
<thead>
<tr>
<th>CORI</th>
<th>Precision at</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.57 2.27 2.99</td>
</tr>
<tr>
<td>10</td>
<td>3.06 4.46 5.89</td>
</tr>
<tr>
<td>20</td>
<td>6.01 8.78 11.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAP</th>
<th>Precision at</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.74 2.30 2.95</td>
</tr>
<tr>
<td>10</td>
<td>3.45 4.57 5.84</td>
</tr>
<tr>
<td>20</td>
<td>6.93 9.17 11.68</td>
</tr>
</tbody>
</table>

Table 1: Precision of the CORI and PCAP strategy, when using the first 1, 2, 4, 8, 16 or 17 clusters. Queries from the fourth week.

\(^3\)Available under a BSD-style license at http://www.seg.rmit.edu.au/zettair/.
the workload, but also to minimize the average response time. Then, we showed that it is possible to find a trade-off between the above two goals by mining usage data from query logs. In fact, it is possible to reduce the number of queries answered by each server by assigning to the same partition those terms that often co-occur together within user queries. It is thus possible to take advantage of the knowledge extracted from past queries in the form of frequent patterns, in order to drive both a partitioning and replication strategy for term-partitioned, large-scale, pipelined WSE.

The term assignment problem. In this novel pipeline approach, a query $Q$ traverses all the servers of $H_\lambda(Q)$ in a pipeline fashion, where $H_\lambda$ is the set of servers containing postings lists for some terms of the query according to the partitioning $\lambda$ of the lexicon. Each server forwards the list of relevant DocIDs, corresponding to the portion of $Q$ processed so far, to the next server of the pipeline.

Since a query may need to traverse several servers, it is important to try to reduce the average number of these servers, in order to improve single-query latency. This can be done, in principle, by carefully allocating terms and associated postings lists to the servers.

In the following we want to give an estimation of the workload incurred by a generic server and the latency of a generic query. We have to state in advance that a WSE is a complex, highly nonlinear environment, where actual query processing times are characterized by high variance and may be very hard to predict [12]. Thus, our modeling analysis can be used to understand the behavior of a WSE, and to identify some average costs incurred.

Given the pair $(t, l_t) \in I$, where $t$ is a term of the lexicon and $l_t$ is the length of its postings list, and we will use the following symbols:

1. $T_{disk}(l_t)$, which models the time to transfer from disk the postings list $l_t$;
2. $T_{compute}(l_t)$, which corresponds to the compute time concerning the postings list $l_t$;
3. $T_{overhead}$, which models the CPU time spent by a server to receive and then send a message.

Let $Q_\lambda$ be the subsets of the terms in $Q$ assigned to the server $j$ according to the partitioning $\lambda$, the time needed by server $j$ to evaluate such query is:

$$T_j^\lambda(Q) = T_{overhead} + \sum_{t \in Q_\lambda^j} (T_{disk}(l_t) + T_{compute}(l_t))$$

We address two different settings. In a disk dominant setting, posting-lists are resident on disk, and therefore $T_{disk}(l_t)$ dominates the cost due to large seek and rotation times. Conversely, if we consider that, nowadays, typical sizes of main memories are huge, and that in a WSE the partitioned index $I$ can benefit from the aggregate main memory of a cluster farm, then we have a network dominant setting where $T_{disk}(l_t)$ is negligible.

Even though each query is served by a single pipeline, multiple queries can be solved concurrently by the system. Moreover, if each server has a sufficient number of distinct sub-queries $Q_j^\lambda$ to answer, the communication time required to transfer queries and partial results among WSE nodes can be overlapped with useful computation.

Under these assumptions:

$$\hat{L}_\lambda(\Phi) = \max_j \sum_{Q \in \Phi} T_j^\lambda(Q)$$

corresponds to the total completion time to answer all the queries in query stream $\Phi$. Thus the following hypothesis holds:

**Hypothesis 1** In a term-partitioned WSE with a partitioning function $\lambda$, the throughput is $O\left(\frac{|\Phi|}{\hat{L}_\lambda}\right)$.

If we were able to balance the server workloads, by finding an appropriate partitioning $\lambda$, we would reduce $\hat{L}_\lambda$, thus improving the throughput of the WSE.

In addition, the time elapsed to process a single query is influenced by the number of servers involved $H_\lambda(Q)$. Devising a partitioning that reduces the average width of queries $w = \sum_{Q \in \Phi} H_\lambda(Q)/|\Phi|$, results in a larger number of terms of the same query processed by the same server, and thus in a reduction of the volume of data transferred over the network.

**Hypothesis 2** In a pipelined term-partitioned WSE with a partitioning function $\lambda$, the average time (latency) for answering a generic query $Q$ is $O\left(\frac{w_j(\Phi)}{N}\right)$.

In a heavily loaded environment, like a large scale Web Search Engine, the throughput is usually the most important quantity to optimize (Hypothesis 1). From the user perspective, however, query response time is the most important figure (Hypothesis 2). Our thesis is that it is possible to find a trade-off, and devise an optimizing technique able to optimize both.

**The Term-Assignment Problem.** Given a weight $\alpha$, $0 \leq \alpha \leq 1$, a query stream $\Phi$, the Term-Assignment Problem asks for finding the partitioning $\lambda$ which minimizes

$$\Omega_\lambda(\Phi) = \alpha \cdot \frac{\varpi_\lambda(\Phi)}{N_\omega} + (1 - \alpha) \cdot \frac{\hat{L}_\lambda(\Phi)}{N_L}$$

where $N_\omega$ and $N_L$ are normalization constants.
Table 2: Percentages of queries as a function of the number of servers involved in their processing.

<table>
<thead>
<tr>
<th>Servers</th>
<th>Baseline Cases</th>
<th>Term Assignment α = 0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>random</td>
<td>bin packing</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>23</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>&gt; 3</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

The term-assignment can be thought as a variation of the classical NP-hard Bin Packing problem, where we have to place objects of certain weights (terms) into a fixed number of different bins (servers) with the aim of optimizing a given objective function (Ωλ). In our case, the objective function not only depends on the weights assigned to terms, but also on the mutual assignment of terms. Intuitively, a very frequent term t may add a considerable load to a server, unless this server already owns a term often co-occurring with t, in which case the load of the server would not increase, and the average width of the queries would be reduced. Therefore, by exploiting a pattern-mining approach, the information about co-occurrence of terms can be used to minimize Ωλ.

In [8], we proposed a greedy algorithm very similar to a greedy bin-packing algorithm without backtracking, which takes into account co-occurrence patterns of terms in the query stream. Our method was compared against a random assignment and against a classical greedy bin-packing algorithm, where the weight of a term is simply its frequency. First, we discovered that there exists a value of α optimizing the trade-off between throughput and response-time. In all our experiments, the optimal value was α = .9. Then, we also measured the quality of our partitioning, as the number of servers required to answer a given query. As reported in Table 2, in all our dataset, our partitioning, increases by almost 50% the number of queries hitting one server only, and it significantly reduces the number of queries requiring more than 3 servers (over a total of 8).

5 Multimedia Queries

With the widespread use of digital cameras, more than 80 billion photographs are estimated to be taken each year, and a significant part of them, over one billion, are published on the Web. In this context, the interest in Web-scale Content-Based Image Retrieval (CBIR) systems to search such huge collections of images is rapidly growing [3]. The content-based search paradigm, based on image similarity computed over the visual features extracted from images, can be used as a valid complement to the consolidated multimedia search paradigm based on textual metadata only. Unfortunately, similarity-based image search is very expensive, so that one of the main research challenge is the scalability of such systems.

We are interested in investigating the adoption of a server-side cache to make CBIR systems more scalable. Such caches enable in fact a shorter average response time, reduce the workload on back-end servers, and improve the overall throughput. They have been shown very effective for text-based Web search engines (WSEs), and query logs constitute the most valuable source of information, not only for evaluating the effectiveness of caching systems storing the results of past WSE queries, but also for improving the cache management policies [5].

Many studies on WSE query logs confirms that users share the same query topics according to an inverse power law distribution. Unfortunately, although there exist several CBIR prototypes, there is no public query log available.

Thus, in order to test a CBIR cache setting, we need to generate a synthetic query log according to a realistic Web-based scenario. Looking at the popularity rank scores on large-scale photo sharing sites [15], this seems to suggest a distribution of topic popularity in the log similar to the one found in text-based query logs. Moreover, we have to take into account the phenomenon of near-duplicates. Given the results of this study, we can estimate at about 8% of the images in the web are near-duplicates.

The CBIR cache we propose is very different from a traditional cache for WSEs, which can be thought as a simple hash table, whose keys are the submitted queries, and the stored values are the associated pages of re-
results, with some policy for replacement of cache entries based on the recency and/or frequency of references. In content-based similarity search, even when the current query objects was never seen in the past, the possible (and measurable) similarity among this query object and the cached objects can be exploited. In other words, we want the cache to be able to return an answer to as many submitted queries as possible: (a) an exact answer when exactly the same query was submitted in the past, and its results were not evicted from the cache; (b) an approximate answer composed of the closest objects currently present cache. In case the quality of the approximated answer is acceptable according to a given measure, the system can return it to the user without querying the underlying content-based index. Our cache exploits the metric property of the distance measure for evaluating the quality of the approximate results.

It is worth considering that, also for near-duplicate variants of a previously cached query objects, we can return a high-quality answer without querying the search engine back-end, thus avoid inserting in the cache a near-duplicate result set associated with the incoming query image. Conversely, a traditional cache, which can answer with exact matches only, would soon be polluted with a lot of very similar result sets.

It is worth noting that the proposed CBIR caching techniques can be used in any scenario in which we need to boost large-scale similarity-based search services for metric objects (e.g., medical data, DNA sequences, financial data).

**Metric cache.** The framework of our CBIR system can be formally defined as follows. Let $D$ be a collection of objects belonging to the universe of the valid objects $U$ and let $d$ be a metric distance function\(^6\), $d: U \times U \Rightarrow \mathbb{R}$, used to measure the similarity between two objects. $(U, d)$ corresponds to a metric space, which is the basic assumption of our work.

The database $D$ can be queried by using two different kind of queries: range queries and $k$ nearest neighbors queries. A range query returns all the objects in the database at distance at most $r$ from a given query object $q \in U$, i.e., $R_D(q, r) = \{ o \in D \mid d(q, o) \leq r \}$. A kNN query returns the $k$ nearest objects to $q$, denoted with $kNN_D(q,k)$. We call $r_q$ the radius of the smallest hypersphere centered in $q$ and containing all its $k$ nearest neighbors in $D$.

We will focus on $kNN$ queries (for some fixed $k$) since such query are more interesting in many similar-

\(^6\)A metric distance has the following properties: non negativity, symmetry, identity, and triangular inequality.
∃qᵢ ∈ C such that rᵢ − d(qᵢ, q) > 0. This means that q is found to be inside the hypersphere centered in qᵢ, with radius rᵢ. In turn, the result of Rᵦ(q, sᵦ) corresponds to the top k', k' = |Rᵦ(q, sᵦ)| ≤ k, nearest neighbors of q in D. The result of such range query can be used to build an approximate result set to the query q, where the top k' objects of the approximate answer are the same of the top k' results of the exact answer.

Figure 5 shows a simple example with objects and queries in a two-dimensional Euclidean space. Intuitively, every cached query qᵢ induces complete knowledge of the metric space up to distance rᵢ from qᵢ. If any subsequent query q is found to be inside the hypersphere centered in qᵢ with radius rᵢ, then, as long as we look inside this hypersphere, we also have complete knowledge of the k', k' ≤ k, nearest neighbors of q.

Supposing that the maximum safe radius sᵦ is not trivial, i.e. sᵦ > 0, we denote with ˜q, the query associated with it:

\[ ˜q = \arg \max_{qᵢ \in C} (rᵢ − d(qᵢ, q)) \]

In order to provide the best possible approximate answer, we need:

1. to find efficiently ˜q among all the cached queries;
2. to choose the additional k − k' objects that are needed to complete the approximate result set.

Discovering ˜q is not straightforward. Consider Figure 5, which shows that, in order to determine the largest safe radius sᵦ, we may need to consider a cached query (qᵢ) that is not the closest one (qₙ) to new incoming query q.

Our QCACHE uses a smaller index built over the query objects q ∈ C to approximate sᵦ, and avoid exploring exhaustively all the queries stored in cache. Using this very small index, QCACHE can select the kₙ cached queries closest to q, and search among them the one that actually maximizes the safe radius (rᵢ − d(qᵢ, q)). Let us denote with ˜sᵦ the resulting approximated value of sᵦ, and with ˜q the corresponding query. It holds that ˜sᵦ is a lower bound of sᵦ.

Given ˜q and ˜sᵦ, we can guarantee that the cached result set kNNᵦ(˜q, k) contains k*, k* ≤ k, of the nearest neighbors of q in the database D. These are the objects o ∈ kNNᵦ(˜q, k) being at distance at most ˜sᵦ from q. Finally, we need to choose a set of k − k* additional objects to produce the approximate answer. Still, we want to avoid searching among all the cached objects, and thus limit the search to the kNN objects close to the kₙ cached queries previously selected.

**Testing the metric cache.** The database of images we used consists in a set of one million objects randomly selected from the CoPhIR collection7. CoPhIR is the largest publicly available collection of high-quality images metadata. Each contains five MPEG-7 visual descriptors (Scalable Color, Color Structure, Color Layout, Edge Histogram, Homogeneous Texture), and other textual information (title, tags, comments, etc.) of about 60 million photos (still increasing) that have been crawled from the flickr photo-sharing site8.

The dissimilarity (or distance) between two images can be evaluated in terms of weighted sum of the distances between each of the five MPEG-7 descriptors used. Moreover, the final distance between two images is metric, according to our assumption.

In order to build a realistic query log for this collection, we first took into consideration the usage information made available by flickr and stored in CoPhIR. For each image, we know the number of times it was seen by any user: the views distribution follows a Zipf-like distribution, which results to be similar to the query topic distribution present in the query logs of textual web search engines [5]. To take into account the phenomenon of the near duplicate images available in the web, we injected a total of 8% of near duplicates9, applying a duplication rate to each image proportional to its popularity, i.e. number of views. Also we divided evenly the popularity of an image among the image itself and its duplicates. Finally, we sampled with replacement 100,000 objects from such collection, where the

---

7CoPhIR stands for COntent-based Photo Image Retrieval, see http://cophir.isti.cnr.it.
8http://www.flickr.com
9Images were obtained by applying random scaling, or cropping, or contrast adjustment, or border addition, or a combination of cropping and contrast adjustment, according to the presence of these alterations measured in [6].
probability of a photo to be selected is proportional to its popularity.

We used a publicly available M-Tree [2] implementation\(^\text{10}\) for indexing the one million images dataset and the queries cached by QCache.

Fig. 6 reports exact, approximate, and total hit ratios as a function of the size \(Sz\) of the cache, measured in terms of the number of stored objects (i.e., the associates features). Assuming that for each cached query only the top-\(k\) results are stored: therefore the cache contains the result sets of \(Sz/k\) queries. Note that some redundancy may occur, whenever an object belongs to the result set of multiple cached queries.

First of all, we note that the number of exact hits is not high, meaning that a traditional cache would not be effective. This is both due to the presence of duplicates, and to the \(\alpha\) value of the Zipfian image views distribution which is smaller than in traditional text-search query logs.

Conversely, the approximate hit ratios achieved are consistently higher. With the largest cache we experimented, it is possible to answer more than one fourth of the queries by storing only one tenth of the database. Since processing an exact similarity query over a metric index has a cost proportional to the size of the indexed collection, this result shows that the proposed caching technique can have an impressive impact on the overall performance of a CBIR system.

We also evaluated the overall quality of the approximate answers, beyond the \(k^*\) guaranteed results, \(k^* \leq k\), of query \(kNN_D(q,k)\). The error introduced is always below 10%, so that we can consider QCache very effective also for approximating similarity queries on the basis on the reuse of past knowledge.

6 Conclusions

In this paper we have shown a number of applications that can benefit from analysis performed on Web Search Engines’ query logs. In particular, the techniques shown have been developed by people at High Performance Computing Laboratory in Pisa. Furthermore, these results have been obtained thanks to the work of many other people. In particular, Diego Puppin formerly a Ph.D. student working in our group and now at Google in Boston; Ricardo Baeza-Yates director of Yahoo! Research Labs Latin America and Barcelona; Domenico Laforenza former lab’s head and now Director of the IIT in Pisa. Other important contributors are: Fabrizio Falchi, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fausto Rabitti.

References


\(^{10}\)http://lsd.fi.muni.cz/trac/mtree/