Answering Keyword Queries through Cached Subqueries in Best Match Retrieval Models

Myron Papadakis[1], Yannis Tzitzikas[1,2]

[1] Computer Science Department, University of Crete, GREECE
[2] Institute of Computer Science, FORTH-ICS, GREECE.
Email: myrpap@csd.uoc.gr, tzitzik@ics.forth.gr

Abstract

Caching is one of the techniques that Information Retrieval Systems (IRS) and Web Search Engines (WSEs) use to reduce processing costs and attain faster response times. In this paper we introduce Top-K SCRC (Set Cover Results Cache), a novel technique for *results caching* which aims at maximizing the utilization of cache. Identical queries are treated as in plain *results caching* (i.e. their evaluation does not require accessing the index), while combinations of cached sub-queries are exploited as in *posting lists* caching, however the exploited subqueries are not necessarily single-word queries. The problem of finding the right set of cached subqueries to answer an incoming query, is actually the *Exact Set Cover* problem. This technique can be applied in any best match retrieval model that is based on a *decomposable scoring function*, and we show that several best-match retrieval models (i.e VSM, Okapi BM25 and hybrid retrieval models) rely on such scoring functions. To increase the capacity (in queries) of the cache only the top-K results of each cached query are stored and we introduce metrics for measuring the accuracy of the composed top-K answer. By analyzing queries submitted to real-world WSEs, we verified that there is a significant proportion of queries whose terms is the result of a union of the terms of other queries. The comparative evaluation over traces of real query sets showed that the Top-K SCRC is on the average *two times faster* than a plain Top-K RC for the same cache size.

Keywords. Information retrieval, Query Processing, Retrieval Models, Ranking, Web search engines, Query log analysis

1 Introduction

Caching is one of the techniques that Information Retrieval Systems (IRS) and Web Search Engines (WSEs) use to reduce processing costs and attain faster response times. Such systems are based on pre-computed indexes for offering fast *word-based* access. An *inverted index* (or inverted file) is a data structure for finding efficiently the documents that contain a particular term. It consists of the *vocabulary* $T = \{t_1, \ldots, t_m\}$ containing all distinct terms extracted from the documents (and optionally their *document frequency*), and the *posting file*. The latter, for each term $t_i \in T$ keeps its *posting list* $I(t_i)$, where each entry of that list contains the identifier of each document that contains t_i, and optionally additional information like the frequency t_i in the document (tf) and the positions of its occurrences. When a query $q = \{t_0, t_1 \ldots, t_n\}$ is submitted, the engine fetches from the disk the inverted lists of the terms $\in q$ and exploits the information stored in these lists to rank and return the matching documents. Ranking is based on scoring functions and in most best match retrieval models (e.g. Okapi BM25, VSM) every document that contains at least a term of q is assigned a positive score indicating its similarity with q. This means that even for a single query the system has to access the index and traverse a large number of postings lists and process a large amount of various data, especially for terms with long posting lists, leading to increased query response time. For a comprehensive study of inverted files for text search engines, see [33].

There are several kinds of information that can be *cached* by an IRS for reasons of efficiency. Caching user queries and their answers, namely *results caching (RC)* [19, 30, 22, 16, 17, 10, 3, 5, 25], is one of the most popular and effective techniques for improving the efficiency of large scale WSEs. Answering a query through a results cache is very fast, since the answer is immediately returned by the cache and query evaluation is not necessary. The major weakness of plain results caching is that it speeds up the query evaluation of the incoming query if and only if the cache contains an *identical* query. Since terms repeat more significantly than queries, an alternative approach is to cache individual query terms and their corresponding posting lists, namely *postings lists caching (PLC)* [22, 4, 17, 5, 2]. Posting lists caching is more advantageous in terms of cache utilization and promises higher hit rates than results caching since cached query terms can be *combined* for answering the incoming query. In practice, the inverted lists of terms form a sort of a smaller in memory inverted file, thus reducing the I/O latency that arises when retrieving the posting lists from the disk. The drawback of posting lists caching is that it does not avoid query evaluation and the required processing can have significant computational cost (especially in best-match retrieval models due to the needs of ranking). Moreover, posting lists require more space in case of large document collections. To compound the weaknesses of the above caching techniques, *multi-level (or hybrid) caches* that cache both query results and query terms have been proposed in order to leverage the advantages of results caching and posting lists caching [22, 4, 17, 25]. Independently of what the cached elements are (query results or posting lists), caching policies can be either static [19, 4], dynamic [19, 22, 16, 4, 17] or hybrid [10].

In this paper we propose and evaluate (through comparative experiments) a new caching scheme for results caching, called SCRC (Set Cover Result Caching). The main idea of SCRC, is to exploit *cached sub-queries* for answering the incoming queries, aiming at maximizing the utilization of the cache.

To grasp the idea Figure 1 shows an example where the query q = "`barack obama facebook`" is submitted to a WSE. In this example, a plain RC (as presented by previous works) reports a cache miss, since there is not an identical cached query and the WSE has to evaluate q from the main index, despite the fact that the terms of the query and their associated answers reside in the cache. In case that a PLC is employed (such as in multi-level caches), it will try to answer q but it will not avoid the query evaluation process (even if it contains all the query terms of the submitted query), as we stated earlier. The technique presented in this paper addresses this problem, specifically the incoming query can be answered by the SCRC by combining the answers of two cached queries; one single word (`facebook`) and one query consisting of two words (`barack obama`).

In this case, the SCRC reports a *"set cover hit"*.

Note that the same cached queries can be exploited for speeding up the evaluation also of "super-queries", e.g. for answering the query "`barack obama facebook nobel prize`" only the posting lists of `nobel` and `prize` have to be fetched from the index. Neither RC nor PLC can offer such speedup.

To the best of our knowledge, none of the previous works on results caching has addressed this problem so far. However, for being successful, one has to devise a technique that:

- employs a fast algorithm for examining if a combination of cached subqueries can be used for answering the incoming query (this is related to the *Exact Set Cover Problem*),
- guarantees that the final composed answer $Ans(q)$ will contain the correct documents and these documents will be ranked correctly (even in result caches that host only the Top-K hits of each cached query), and
- offers performance gains in realistic query streams.

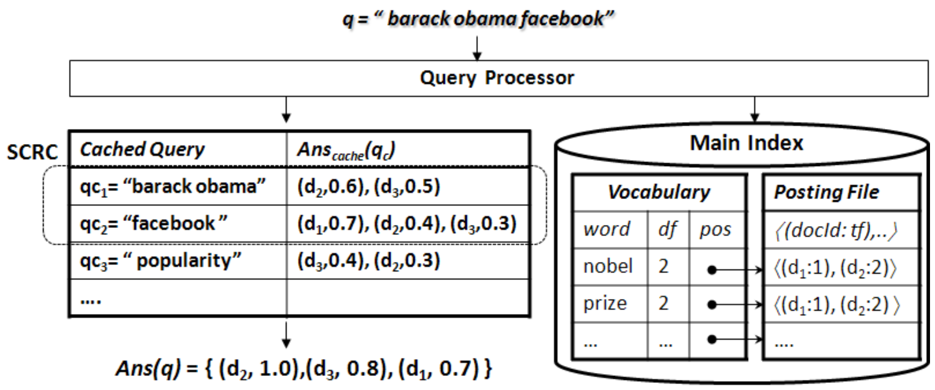

q = " barack obama facebook"
Query Processor
SCRC
Cached Query
Ans_cache(q_c)
qc1= "barack obama" | (d2,0.6), (d3,0.5)
qc2= "facebook" | (d1,0.7), (d2,0.4), (d3,0.3)
qc3= " popularity" | (d3,0.4), (d2,0.3)
....
Ans(q) = { (d2, 1.0),(d3, 0.8), (d1, 0.7) }
Main Index
Vocabulary
word df pos
nobel 2
prize 2
Posting File
⟨(docId: tf),..⟩
⟨(d1:1), (d2:2)⟩
⟨(d1:1), (d2:2)⟩
....

Figure 1: Example of answering a query by combining the answers of cached subqueries

Since identical queries are treated as in plain results caching (i.e. their evaluation does not require accessing the index, nor any kind of postprocessing), while combinations of cached sub-queries are exploited as in posting lists caching (which can be considered as a special case of our approach since our subqueries are not necessarily single-word queries), we can say that

SCRC generalizes results caching and posting lists caching and provides a more general solution. Moreover, the structure of SCRC is the same with that of RC, meaning that there is not any additional storage overhead.

Our work is different from what has been proposed in past in the sense that SCRC can preserve the final ranking of the documents, and thus it can be used in best match retrieval models (e.g. Okapi BM25, VSM). In general, it is applicable to ranking methods based on *decomposable* scoring functions. It can also be applied to more complex scoring functions (that modern WSEs use and include both a query-dependent score and a query-independent score). It can also be easily adjusted to comply with the semantics (i.e *AND*, *OR* semantics) of the underlying WSE.

To further increase the utilization of the cache, it can also store only the top-K results of each cached query, and we show how the query evaluation algorithm can express and measure the accuracy of the composed top-K answer using two metrics (exact top-K' part, and correct relative order top-K'' part).

SCRC has been evaluated experimentally over two kinds of indexes: one Inverted File-based and one DBMS-based. For obtaining meaningful and representative results, our evaluation is based over query sets submitted to real world WSEs (Altavista, Excite). In addition we introduce metrics suitable for identifying those characteristics of a query stream (e.g. average query length, identical query ratio, partial exact set cover density, set cover density) that affect the speed up obtained by the SCRC.

In a nutshell, the key contributions of our work are:

- the notion of decomposable scoring functions and the presentation of several classical best-match retrieval models (like VSM and Okapi BM25) as well as hybrid retrieval models that rely on such scoring functions,
- a query evaluation scheme that is based on cached subqueries and the identification of the conditions that a retrieval model (scoring function) should satisfy for adopting this method (decomposability),
- an analysis of real world query traces which shows that a high percentage of queries can be answered as a combination of the terms of other submitted queries,
- a variation of the method where only the top-K answers are stored in the cache and metrics for measuring the accuracy of answers derived through set cover hits, and
- comparative experimental results over real query sets that show that a Top-K SCRC is on the average 2 times faster than a Top-K RC for the same cache size.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces and elaborates on decomposability. Section 4 explains how the set cover problem relates to query evaluation. Section 5 focuses on the algorithmic perspective. Section 6 elaborates on the quality of the composed top-K answers. Section 7 reports extensive comparative experimental results, and finally, Section 8 concludes the paper. Extra material (proofs, experimental results and examples) are given in the appendix.

2 Related Work

Result Caches. A large body of work has been devoted to *results caching* [19, 29, 22, 16, 17, 10, 3, 5, 25]. In one of the earliest works, Markatos [19] introduces caching query results as a means to reduce the response time of a search engine. He showed the existence of high temporal locality in queries and demonstrated that warm, large caches of search results can attain hit ratios of close to 30%. Xie and ÓHallaron [29] also showed that queries have significant locality, with query frequency following a Zipf distribution.

Lempel and Moran [16] proposed a probabilistic caching algorithm for results caches that estimates the number of the result pages that need to be cached for a submitted query. An hybrid results cache, referred as *SDC cache* is proposed by Fagni et al. [10] , where the available cache space is split into a static and a dynamic cache. Experimental results showed that devoting most of the cache space to the static part along with prefetching yields the highest hit rates. Further improvements on the SDC policy are discussed by B.Yates et al.[1] and Ma and Wang [18].

Posting Lists Caches. A large body of work has also been devoted to *posting lists caching* [22, 4, 17, 5, 32, 2]. B.Yates et al. [5, 2] explored the impact of different static and dynamic techniques for posting lists caching. They showed that posting lists caching offers higher hit ratios than results caching and proposed a new static caching algorithm for posting lists. Zhang et al. [32] evaluated several posting lists caching policies and addressed the impact of compression on caching efficiency.

Multi-level Caches. There are several works [22, 4, 17] that propose two or more levels of caching and mix ideas both from results and posting list caching. A *two-level caching* system was firstly proposed by Saraiva et al. [22] for caching query results and posting lists in a dynamic setting. A similar cache organization is proposed by B.Yates and Jean [4] with a RC and a PLC but they consider static caches, holding the most popular queries and terms respectively. Long and Suel [17] propose a three-level caching architecture that includes an intermediate level with on-disk caching of the intersections of the posting lists of frequently occurring pairs of terms.

Figure 2 shows some criteria that discriminate the caching approaches that we discussed earlier.

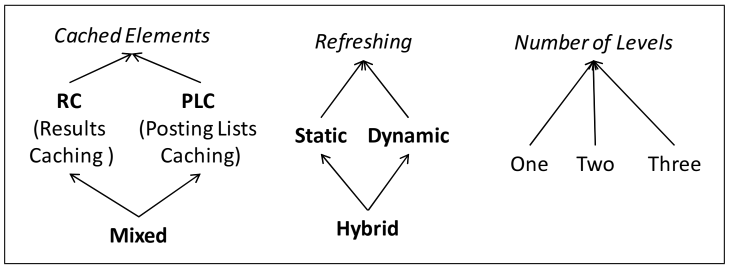


Figure 2: Caching approaches discrimination criteria

SCRC. The SCRC that we introduce in this paper, can be considered as a significant advancement of Results Caching (RC). Our work is different from Long and Suel [17], since we store query results and not intersections of posting lists. Note that the intersection of posting lists may not include some documents that contain only one of the query terms (and note that such documents are contained in the answer of best match retrieval models). Moreover, the documents in the list are not ranked.
We should stress that ranking is of paramount importance in modern WSEs and IRS. Most of the previous works dealing with posting lists caching use various techniques to determine the top-k results of a query (i.e by presorting the lists according to their contributions to the score and terminating the traversal early or by removing low-scoring postings from the index altogether), since lists require considerable space to be stored fully and long lists also compensate the benefits of PLC. However, none of these works reports experimentally accurate results for the accuracy/quality of the composed top-K answers, as we do. More specifically, Saraiva et al. [22] process the inverted lists of the PLC in most cases partially but do not report results for the accuracy of their composed answers (for those queries that are answered by the PLC). Long and Suel [17] support theoretically that their approach could be adapted to pruned schemes but in their experiments they report that they use a full traversal of the list intersections.

As regards, cache filling and refreshing, SCRC could be used in both static, dynamic or hybrid refreshing policies, in general we believe that the various complex caching policies (as regards refreshing or layering) which include a RC, can benefit by replacing their RC by a SCRC, therefore we expect gains in various settings.

3 Decomposable Scoring Functions

Here we introduce the notion of *decomposability* which is fundamental for the applicability of the proposed caching technique. For convenience, all notations used in this section are defined also in Table 1 (the reader can refer to this only when needed).

Let $Ans(q) = \{\ d \in Obj \ \mid \ Score(d, q) > 0\}$ be the answer of a query q where $Score(d, q)$ quantifies the relevance of d to q, or/and the importance of d (e.g. PageRank score).

Def 3.1 (Decomposability)
Let $q = \{t_0, t_1, \ldots, t_n\}$ be a query of n distinct terms. A scoring function $Score : Obj \times Q \rightarrow \Re$ is decomposable *iff its value (the score of a document d w.r.t to q) is the sum of the individual term relevances, i.e.:*

$$Score(d, q) = \sum_{t \in t(q)} Score(d, t)$$

This means that if $Score$ is decomposable, then for every document d it holds:
$Score(d, \text{“}\texttt{barack obama facebook}\text{”}) = Score(d, \text{“}\texttt{barack}\text{”}) + Score(d, \text{“}\texttt{obama}\text{”}) + Score(d, \text{“}\texttt{facebook}\text{”})$.

Given a universe of elements $U = \{u_1, u_2, \ldots, u_n\}$ and a family $C = \{S_1, \ldots, S_k\}$ of nonempty subsets of U, we call C a *partition* of U, if $\bigcup_{S_i \in C} S_i = U$ and $S_i \cap S_j = \emptyset$ for every $i \neq j$. It follows easily by induction, that if C is a partition of $t(q)$, then

$$Score(d, q) = \sum_{q_c \in C} Score(d, q_c) \tag{1}$$

i.e. the score of a document d with respect to a query q containing all terms in U is the sum of the scores of d in the answers of the queries that form the family C. In our example this means that $Score(d, \text{“}\texttt{barack obama facebook}\text{”}) = Score(d, \text{“}\texttt{barack obama}\text{”}) + Score(d, \text{“}\texttt{facebook}\text{”})$.

Table 1: Notations

Symbol	Meaning
T	the vocabulary of the inverted index
Obj	the set of all potential document identifiers (docIds)
I	the inverted index, $I = \{I(t_0), I(t_1), \ldots, I(t_n)\}$
$I(t_i)$	the inverted list of term $t_i \in T$
$tf_{d,t}$	frequency of term t in document d
$df(t)$	the document frequency of term t
$idf(t)$	the inverse document frequency of term t
$w_{d,t}$	weight of term t in document d
$w_{q,t}$	weight of term t in query q
$W(d)$	norm of document vector d
$W(q)$	norm of query vector q
Q_e	a stream of queries
q	a query submitted to the search engine, $q \in Q_e$
$t(d)$	the set of the distinct words of document d
$t(q)$	the set of the distinct words of the query q
$g(d)$	static query-independent score of document d
$pr(d)$	the pagerank score of document d
$Score(d, q)$	the score of document d w.r.t query q
$Sim(d, q)$	the similarity score of document d with respect to q
$Sim_{BM25}(d, q)$	the BM25 score of a document d with respect to q
$Ans(q)$	the answer of the WSE for the query q
Cache Notations	
M	the available cache memory
Q_c	all the cached queries
q_c	a query stored in the cache, $q_c \in Qc$
$Ans_{cache}(q_c)$	the answer of the cache for the query q_c

3.1 Examples of Decomposable Scoring Functions

Now we will show that several best-match retrieval models rely on decomposable scoring functions.

Okapi BM25 model: Several $tf \cdot idf$ based scoring models that are widely used in the literature (e.g. in [26, 7]) use decomposable scoring functions for ranking the relevant documents. For example, the Okapi BM25 model uses the following scoring function:

$$Sim_{BM25}(d, q) = \sum_{t \in t(q)} idf(t) * \frac{tf_{d,t} * (k_1 + 1)}{tf_{d,t} + k_1 * (1 - b + b * \frac{|d|}{avgdl})}$$

where $tf_{d,t}$ is the frequency of the term t in the document d, $|d|$ is the length of the document d in words, $avgdl$ is the average document length in the text collection from which documents are drawn, and k_1 and b are free parameters, usually chosen as $k_1 = 2.0$ and $b = 0.75$. The inverse document frequency $idf(t)$ of the query term t is computed as $idf(t) = \log \frac{N - df(t) + 0.5}{df(t) + 0.5}$ where N is the total number of documents in the collection.

It is not hard to see that the scoring function of Okapi BM25 is *decomposable*, i.e. it satisfies the condition of Def. 3.1. This is true because each addend corresponding to one query term, does not contain any quantity that depends on a different query term. The proof is trivial:

$$
\begin{aligned}
Sim_{BM25}(d,q) &= \sum_{t \in t(q)} idf(t) * \frac{tf_{d,t} * (k_1+1)}{tf_{d,t} + k_1 * (1 - b + b * \frac{|d|}{avgdl})} \\
&= idf(t_1) * \frac{tf_{d,t_1} * (k_1+1)}{tf_{d,t_1} + k_1 * (1 - b + b * \frac{|d|}{avgdl})} + \\
&\quad idf(t_2) * \frac{tf_{d,t_2} * (k_1+1)}{tf_{d,t_2} + k_1 * (1 - b + b * \frac{|d|}{avgdl})} + \\
&\quad \ldots + \\
&\quad idf(t_n) * \frac{tf_{d,t_n} * (k_1+1)}{tf_{d,t_n} + k_1 * (1 - b + b * \frac{|d|}{avgdl})} \\
&= Sim_{BM25}(d,t_1) + Sim_{BM25}(d,t_2) + \ldots + Sim_{BM25}(d,t_n) \\
&= \sum_{t \in t(q)} Sim_{BM25}(d,t)
\end{aligned}
$$

The Vector Space Model: According to VSM, the relevance between a document d and a query q, denoted by $Sim_{cos}(d,q)$, equals the cosine of the angle of their vectors:

$$Sim_{cos}(d,q) \quad = \quad \frac{\sum_{t \in t(d) \cap t(q)} w_{d,t} \cdot w_{q,t}}{W_q \cdot W_d}$$

where $w_{x,t}$ is the weight of word t in document or query x (i.e. $w_{x,t} = tf_{x,t} \cdot \log(\frac{N}{df(t)})$, while W_q and W_d denote the *Euclidean* norms of the query q and document d vectors respectively, i.e. $W_q = \sqrt{\sum_{t \in t(q)} w_{q,t}^2}$ and $W_d = \sqrt{\sum_{t \in t(d)} w_{d,t}^2}$.

This function is *not decomposable*, since Def. 3.1 does not hold, i.e. $Sim_{cos}(d,q) \neq \sum_{t \in t(q)} Sim_{cos}(d,t)$. However, the term W_q in the denominator can be omitted since it does not affect the relative ordering of the documents, nor the retrieval effectiveness [33]. Let $Sim'_{cos}(d,q)$ denote the modified version:

$$Sim'_{cos}(d,q) = \sum_{t \in t(d) \cap t(q)} \left(\frac{w_{d,t} \cdot w_{q,t}}{W_d} \right)$$

This scoring function is *decomposable*, because the denominator does not depend on any query term, and thus it holds:

$$Sim'_{cos}(d,q) = \sum_{t \in t(q)} Sim'_{cos}(d,t)$$

Hybrid Ranking: In modern WSEs, the final score $Score(d,q)$ usually depends on both the *query-dependent* score (similarity score) and a *query-independent* score. The query-independent score $g(d)$ of a document d is estimated by using various techniques (e.g. link analysis or query log analysis) which quantify the importance of d, and this is done off-line.

We shall call such scoring functions *hybrid*, and they usually have the form:

$$Score(d,q) = a * g(d) + (1-a) * Sim(d,q) \tag{2}$$

where $0 < a < 1$. The query-independent factors $g(\cdot)$ are precomputed offline and are stored either in the main memory or in the disk if the main memory is not enough.

This scoring function is what we call *semi-decomposable*. We call a scoring function *semi-decomposable*, if it holds:

$$Score(d,q) = \otimes(f(d), \sum_{t \in t(q)} Score(d,t))$$

where $f(d)$ is a value based on d, and $\otimes$ stands for one or more arithmetic operations. The evaluation of such scoring functions requires a post-processing and for getting the value of $f(d)$ we may have to fetch information from the index. It follows from the form of Eq. (2), that if $Sim()$ is decomposable then $Score(d,q)$ is semi-decomposable (here $\otimes(f(d), Score()) = a * g(d) + (1-a) * Score()$).

As another example, in the context of the Mitos WSE [21], we have developed an hybrid retrieval model, which is a combination of boolean plus similarity ranking (i.e Okapi BM25, VSM) and favors documents that contain all the terms of the query q (but does not exclude documents that do not contain all the terms of q). The hybrid score of a document d w.r.t q, denoted by $Score_{Hybrid}(d, q)$ is computed by:

$$Score_{Hybrid}(d, q) = Sim(d, q) + freq(d, q) \tag{3}$$

where $freq(d, q)$ is the number of terms of q that document d contains. Clearly, if we want to include in $Ans(q)$ only those documents containing all the query terms of q, then we consider only those documents whose score is greater than or equal to $|t(q)|$. Notice that if $|t(d) \cap t(q)| < |t(d') \cap t(q)|$, then $Score_{Hybrid}(d, q) < Score_{Hybrid}(d', q)$, since $Sim(d, q) \in [0, 1]$. It follows easily that if $Sim()$ is decomposable, then this scoring function is also decomposable, since the condition of Def. 3.1 holds. If $Sim()$ is semi-decomposable, then this hybrid scoring is also semi-decomposable.

Finally, we should note that even if the score is not a linear combination of the query-dependent score and the query-independent score, the scoring function can be *decomposable*, e.g. this holds for the scoring function described at [25].

4 Set Cover Problem

In this section we discuss the *set cover problem* as well as variations of the problem which are required for introducing the cache-based query evaluation method.

Consider a cache which for each cached query string it stores the id and the relevance score of each document in the answer of the query. Table 2 shows the cache structure, which is the same as the structure of a RC [1]. Let Q_c denote the queries that have been cached, and let q be the incoming query. Let first define what we call *lower* queries (else "subqueries").

Table 2: SCRC Cache Structure

SCRC Cache structure	
Q_c	**Answer**
q_{c_0}	$Ans_{cache}(q_{c_0}) = \{(d_i, Score(d_i, q_{c_0})), \ldots\}$
$\ldots$	$\ldots$
q_{c_n}	$Ans_{cache}(q_{c_n}) = \{(d_i, Score(d_i, q_{c_n})), \ldots\}$

Def 4.1 (Lower Query) *A query q_c is a lower query of q iff $t(q_c) \subset t(q)$. We can define the set of cached lower queries as $Lower(q) = \{q_c \in Q_c \mid t(q_c) \subset t(q)\}$.*

This means that *any* cached subset of the terms of q is a lower query. For example if $q = \{a, b, c\}$ and $q_c = \{c\}$ then $q_c \in Lower(q)$.

It is not hard to see, that if a family C of sets in $Lower(q)$ covers all elements of q, then we can compute the answer of q by combining the answers of these cached queries. If furthermore C is a *partition of* $t(q)$, then we can compute the correct scores (according to Eq. (1)).

Moreover, one might argue that it is desirable to reuse the answers of a small (ideally the smallest) set of cached queries (for reducing the number of union operations and the summation of partial scores). This drives us to the *Set Cover Problem* [8, 28].

Def 4.2 (Set Cover Problem)
Given a universe $U = \{u_1, u_2, \ldots, u_n\}$ of elements and a family $F = \{S_1, \ldots, S_k\}$ of subsets of U where $\bigcup_{S_i \in F} S_i = U$, a set cover *is the minimum in size subfamily C of F such that $\bigcup_{S_i \in C} S_i = U$.*

Let describe through an example, how the set cover problem is related to our objective. Consider a cache containing the following four queries: $\{a, b\}$, $\{b\}$, $\{c\}$ and $\{d, e\}$. Now suppose that a query with the terms $\{a, b, c\}$ arrives. This situation is

[1] Regarding *semi-decomposable* scoring functions, the SCRC structure does not need to change at all. The score of each document in a cached answer is its decomposable query-dependent score.

modeled as follows:
$U = t(q) = \{a, b, c\}$
$F = \{\{a, b\}, \{b\}, \{c\}\}$
$Lower(q) = \{\{a, b\}, \{b\}, \{c\}\}$
The solution to the *exact set cover* is:
$C = \{\{a, b\}, \{c\}\}$.

However a set cover (according to Def. 4.2) *is not necessarily a partition* (although it happened to be a partition in the previous example). To stress disjointness (which is important for computing scores correctly), we introduce what we call *exact set covers*.

Def 4.3 (Exact Set Cover)
A family of sets $C = \{S_1, \ldots, S_k\}$ is an exact set cover *of U, for short* ESC*, if $\cup_{S_i \in C} = U$ and $S_i \cap S_j = \emptyset$ for every $i \neq j$.*

In other words, it is a *partition*. We can use Figure 3 to clarify the difference between a plain set cover and an exact set cover. The figure shows a family of sets $F = \{S_1, S_2, S_3, S_4\}$ each being a subset of $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Here the *set cover* is the family $C = \{S_1, S_2\}$, since $S_1 \cup S_2 = U$ and $|F|$ is minimal. However, C is *not an exact set cover*, since S_1 and S_2 are not disjoint. An *exact set cover* is the family $C' = \{S_1, S_3, S_4\}$, since $S_1 \cup S_3 \cup S_4 = U$ and $S_1 \cap S_3 = \emptyset$, $S_1 \cap S_4 = \emptyset$, and $S_3 \cap S_4 = \emptyset$.

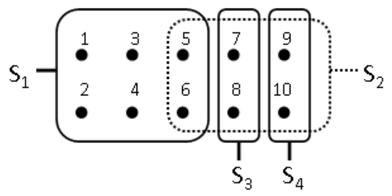


Figure 3: Plain Set Cover Vs Exact Set Cover

We are interested in *exact* set covers, and not just plain set covers, in order to compute the scores of the documents correctly (recall Equation 1). In a plain set cover a term may occur *more than once*, and as a result the scores of some documents *will not be correct*. With an exact set cover, no term occurs more than once, consequently all scores are correct. If however a system provides as answers unranked documents (computed by exact retrieval models) or documents ranked in a *query-independent* way (e.g. PageRank), then a plain set cover is enough. An example demonstrating why a plain set cover would lead to a wrong computation of the scores of the documents in the final answer $Ans(q)$ of a query q in a best-match retrieval model, is given in the Appendix (Example D.1).

Notice that the formulation of the exact set cover problem does not have any kind of minimality constraint. The exact set cover problem is actually a decision problem (returning YES if an exact cover exists, and NO otherwise) and it is NP-complete [15].

We shall now introduce the notion of a *partial exact set cover* which is needed for tackling the cases where not all terms of $t(q)$ can be covered by the cached queries.

Def 4.4 (Partial Exact Set Cover)
Given a universe $U = \{u_1, u_2, \ldots, u_n\}$ of elements and a family $F = \{S_1, \ldots, S_k\}$ of subsets of U such that $\bigcup_{S_i \in F} S_i = U$, a partial exact set cover *of U, or short* PESC*, is a subfamily C of F that is a partition of a proper subset of U (i.e. it contains pairwise disjoint sets and $\bigcup_{S_i \in C} S_i \subset U$), and there is no element $X \in F \setminus C$, such that $C \cup X$ is again a partition of a proper subset of U.*

For instance, if $U = \{1, 2, 3, 4\}$ and $F = \{\{1, 2\}, \{2, 3\}, \{4\}\}$ then we have two partial exact set covers: $\{\{1, 2\}, \{4\}\}$ and $\{\{2, 3\}, \{4\}\}$.

It follows that a *PESC C* can be *extended* (i.e. we can add to it a new set of terms) to become an exact set cover.

Returning to the problem at hand, if C is a *PESC* and $Cterms = \cup_{S_i \in C} S_i$, then we shall use $Rem(q)$ to denote the remainder terms of q, i.e. $Rem(q) = t(q) \setminus Cterms$. This means that if C is a *PESC* consisting of cached queries, we can compute the answer of q by accessing the main index in order to retrieve the posting lists of the terms in $Rem(q)$. We should also note that $|Rem(q)|$ can be used for quantifying the quality of a PESC, i.e. the less this quantity is, the better.

Overall we could say that the objective is to "approximate" as better as possible the set of terms of the submitted query by cached queries. For the problem at hand, we can distinguish various kinds of such approximations: a) exact ones (this category contains what we call identical queries and exact set covers), b) not exact ones (meaning that some terms of the incoming query are not covered), c) valid ones for best match retrieval models (i.e. those that enable computing the scores of the documents correctly), and d) valid only for exact match retrieval models (i.e. for answers without ranking).

5 Algorithms

In this section we provide and discuss algorithms (a) for finding the *cached subqueries* of an incoming query q, (b) for checking if the found subqueries can form an *exact set cover* (or a *partial exact set cover*), and (c) for exploiting the cached answers of these subqueries (and the index if required) for computing the answer of q.

5.1 Finding Cached SubQueries

Consider an incoming query q and a cache hosting a set of queries Q_c. For finding the *lower* queries of q we can follow two main approaches: the *scan*-based and the *hash*-based. According to the *scan*-based we scan the entire Q_c, in order to examine if each cached query is a *lower* query of the incoming query, so if $|Q_c|$ is high then this technique turns out inefficient. An alternative approach (the one we used in our experiments) is to have a hash-based cache and to perform $2^{|t(q)|} - 2$ lookups, i.e. one for every possible subset of $t(q)$ (we subtract 2 due to q itself and the empty set). This approach is faster than the scan-based if $|Q_c|$ is high and $|t(q)|$ is low (e.g. for $|t(q)| = 3$ it requires 6 lookups). More precisely, we choose the hash-based approach if the required lookups are less than $|Q_c|$, i.e. if $|t(q)| < \log(|Q_c| + 2)$ (e.g. if $|Q_c| = 30$, then the hash-based approach is preferable when $|t(q)| < 5$).

We compared these approaches using a trace of queries submitted to the Excite WSE, where $|Q_e| = 37,096$. At first we grouped the queries according to their length. We created a cache hosting 10^4 queries (i.e. $|Q_c| = 10^4$) and we examined the efficiency of the above approaches by looking up and fetching the *lower* queries of each query. Each approach was tested against queries composed of X or more terms, $1 \leq X \leq 10$. Table 3 shows the average execution times for finding the lower queries of a query, i.e column $X = 1$ shows the average execution time of both approaches for those queries $q \in Q_e$ where $|t(q)| \geq 1$. Regarding the hash-based approach the execution times include also the cost for generating the subsets of q and the look-up in the cache.

Table 3: Strategies for finding lower queries

	$t(q) \geq X$									
	$X=1$	$X=2$	$X=3$	$X=4$	$X=5$	$X=6$	$X=7$	$X=8$	$X=9$	$X=10$
Queries ($\|Q_e\|$)	37,096	24,814	9,819	3,100	930	308	118	47	21	5
Approach	**Average Time to find lower queries (ms)**									
Hash Based	**0.027**	**0.051**	**0.105**	**0.261**	**0.654**	**1.57**	**3.57**	**7.97**	**15.6**	**56.2**
Scan-based	90.4	119.24	153.64	198.71	192.84	210.2	229.60	244.0	264.0	309.4

We observe that the hash-based approach substantially outperforms in all cases the scan-based approach (shown in bold in the table). Regarding the hash-based approach we observe that its execution time increases as the number of the query terms increases. However the majority of user queries contain less than 5 words ($|t(q)| < 5$).

5.2 Finding (Partial) Exact Set Covers

We have just seen how we can find the *lower* queries of q. Now we have to check if they can form an exact set cover of $t(q)$. Note that set-covering is an NP-Hard problem. A *greedy* set cover algorithm which achieves logarithmic approximation solution is described in [8]. Below we shall introduce a variation of this algorithm for finding *exact* set covers. It is named `EfficientGreedyExactSetCover`, and is given in Algorithm 1. The algorithm is greedy: for covering the set $t(q)$ at each stage it chooses the lower query which contains the largest number of uncovered elements.

Regarding its computational cost, let us start from the time required to produce the *lower queries* of the incoming query q. This can be done using Alg. `Lower(Qc, q)` (Alg. 2). The maximum number iterations of this algorithm is $\min(2^{|t(q)|} - 2, |Q_c|)$, since the set of the lower queries of q can be computed either by a hash-based lookup method, or by scanning the entire Q_c. Recall that in this algorithm Q_c is implemented as a hash data structure.

Algorithm 1 `EfficientGreedyExactSetCover`($\mathtt{Q_c}, \mathtt{q}$)

```
1:  $C \leftarrow \emptyset$; $U \leftarrow t(q)$; $Visited \leftarrow \emptyset$
2:  $F \leftarrow$ Lower($\mathtt{Q_c}, \mathtt{q}$)                    ▷ $F$ is the set of lower queries
3:  while $U \neq \emptyset$ do
4:      $S \leftarrow F.removeMax()$                    ▷ $S$ is the top-node of the heap
5:      if (($S \neq null$) and ($S \cap Visited \neq \emptyset$)) then    ▷ If $S$ cannot lead to an exact set cover
6:          continue                    ▷ Skip this iteration
7:      end if
8:      if ($S = null$) then                    ▷ the heap is empty
9:          if ($C = \emptyset$) then
10:             return $(null, NOTHING)$                    ▷ Nothing found
11:         else
12:             return $(C, PESC)$                    ▷ Partial Exact Set Cover found
13:         end if
14:     else                    ▷ the addition of $S$ could lead to an exact set cover
15:         $C \leftarrow C \cup \{S\}$                    ▷ $S$ is added to the formed solution
16:         $U \leftarrow U - S$                    ▷ $U$ now contains only the uncovered elements
17:         $Visited \leftarrow Visited \cup S$                    ▷ $Visited$ contains all terms of the elements of $C$
18:     end if
19: end while
20: return $(C, ESC)$                    ▷ Exact Set Cover found
```

Algorithm 2 `Lower`($\mathtt{Q_c}, \mathtt{q}$)

```
1:  $F \leftarrow \emptyset$;                    ▷ $F$ is a max priority heap
2:  if ( $|t(q)| < \log(|Q_c| + 2)$ ) then                    ▷ Hash-based lookup
3:      $SubQueries \leftarrow P(q) \setminus \{t(q), \{\}\}$                    ▷ $P(q)$: Powerset of $q$
4:      for (each $P_i \in SubQueries$) do
5:          if ($P_i \in Q_c$) then                    ▷ Access through Hash
6:              $F.insert(|P_i|, P_i)$                    ▷ Insert $P_i$ in the heap
7:          end if
8:      end for
9:  else                    ▷ Scan-based lookup
10:     for each $q_c \in Q_c$ do
11:         if ($q_c \subset q$) then
12:             $F.insert(|t(q_c)|, q_c)$                    ▷ Insert $q_c$ in the heap
13:         end if
14:     end for
15: end if
16: return $F$
```

By using a *priority queue* (implemented as a max heap) for storing the lower queries (i.e. F), the insertion of each $F_i \in F$ requires $\mathcal{O}(\log |F|)$ time, where $|F| \leq 2^{|t(q)|} - 2$.

Although the maximum iterations are $\min(2^{|t(q)|} - 2, |Q_c|)$, the insertions in the heap are $|F|$ (since there are maximum $|F|$ lower queries), so the time complexity of Algorithm 2 is $\mathcal{O}(\min(2^{|t(q)|} - 2, |Q_c|) * \log |F|)$.

Alg. $\mathtt{Lower(Q_c, q)}$ (Alg. 2) is invoked in line 2 of Alg. 1. The main loop of Alg. 1, is executed at most $|t(q)|$ times, where $F.removeMax()$ is invoked at each iteration in order to find the $F_i \in F$ that maximizes $|F_i \cap U|$. To find and remove the F_i that maximizes the above expression, we just have to remove the root element from the heap and finding the max element in the heap requires $\mathcal{O}(1)$ time. However for removing it, and thus keep up-to-date the heap, requires $\mathcal{O}(\log |F|)$ time. So the cost of the main loop is $\mathcal{O}(|t(q)| \log |F|)$.

It follows that the total time complexity of Algorithm 1 is:

$$\mathcal{O}(\log |F| * (|t(q)| + \min(|F|, |Q_c|)))$$

where $|t(q)|$ is in practice less than 5 and $|F| \leq 2^{|t(q)|} - 2$.

This algorithm returns either an *exact* set cover or a *partial* exact set cover (the latter *is not a set cover*).

If an *exact* set cover is returned, then we need to merge (and process accordingly) the answers of the sub-queries belonging to the set cover to produce the final answer.

If a *partial* exact set cover is returned, then we also have to access the main index in order to retrieve the posting lists of the missing terms. We shall use $Rem(q)$ to denote the remainder of q, where $Rem(q) = t(q) \backslash Cterms$, where $Cterms$ denotes the terms of the returned cached queries. Clearly, if C is an *exact* set cover, then $Rem(q) = \emptyset$.

In the sequel we shall use the abbreviation $q = ESC(qset)$ (where $q \notin qset$) to denote that $qset$ is an *exact* set cover of q found by the greedy algorithm. With $q = PESC(qset)$ we will denote that $qset$ is a *partial* exact set cover of q (found by the greedy algorithm) and that no exact set cover was found.

Approximation: Experimental Results

According to the experimental results that will be presented at Section 7.1 the exhaustive (non-greedy algorithm) returned on average only 0.2% - 0.6% more exact set covers than Algorithm 1.

Some theoretically derived bounds for some cases are discussed in Appendix B.

5.3 Query Evaluation

If the SCRC cache contains an identical cached query (lines 1,2 of Alg. 3), then it will return the answer of this query in the same way that a RC would do. Now, let C be the subqueries returned by $\mathtt{EfficientGreedyExactSetCover(q, Q_c)}$. If C is an exact set cover, then the answer of q can be computed based only the answers of the subqueries in C. If C is a partial exact set cover then we have to forward the remainder of the query $Rem(q)$ to the main index in order to construct the final answer of q. If C is null, then q must be forwarded to the main index and it will be evaluated from the scratch. The complete evaluation process for retrieval models using decomposable scoring functions is described by Algorithm 3.

Regarding semi-decomposable scoring functions only line 19 of Algorithm 3 needs to change, since in order to assign the final score to each matching document d w.r.t a query q, we must post-process the $Score(d, q)$, as computed by the cache. Note that the post-processing operands and operations depend on the form of the scoring function, as we described in Section 3.

Furthermore, we have to note that if we want to force *AND* semantics (i.e get only those documents containing all the terms of the incoming query q as perhaps some commercial WSEs do), we do not have to change the SCRC structure. In this case, a document d will be included in $Ans_{cache}(q_c)$ of a cached query q_c iff d contains all the query terms of q_c. Consequently, if C is an ESC of the terms of q, then d will be included in $Ans(q)$, iff d appears in all the cached answers of C. Regarding the query evaluation algorithm, we just have to replace the *union* operation with an *intersection* operation (lines 9, 12 and 16 of Alg. 3) This variation does not affect the query evaluation cost, since in the first case we retain in $Ans(q)$ only *once* those documents appearing in more than one cached answers

Algorithm 3 `getAnswer(Qc, q)`

```
1: if ($Q_c.get(q) \neq null$) then                                   ▷ if $q$ is cached (identical hit)
2:     return $Q_c.get(q).answer$                                    ▷ return the cached answer
3: else
4:     $(C, note) \leftarrow$ EfficientGreedyExactSetCover(q, Qc)     ▷ Call to Alg. 2
5:     if ($note = null$) then                                       ▷ no subquery is found
6:         return null                                 ▷ $q$ should be answered using the main index
7:     end if
8:     if ($note = ESC$) then                                        ▷ Exact set cover found
9:         $A \leftarrow \bigcup_{q_c \in C} Ans_{cache}(q_c)$        ▷ the union of the hits of the subqueries in C
10:    end if
11:    if ($note = PESC$) then                                       ▷ Partial exact set cover
12:        $PartialAnswer \leftarrow \bigcup_{q_c \in C} Ans_{cache}(q_c)$
13:        $CTerms \leftarrow \bigcup_{q_c \in C} t(q_c)$             ▷ The terms of $q$ that are covered by $C$
14:        $Rem(q) \leftarrow t(q) \setminus CTerms$                  ▷ The terms of $q$ that are not covered by $C$
15:        $C \leftarrow C \cup \{Rem(q)\}$          ▷ Added to $C$ for considering it during the computation of scores
16:        $A \leftarrow PartialAnswer \cup Ans(Rem(q))$              ▷ The hits of the answer of $q$
17:    end if
18:    for each $d \in A$ do                                         ▷ Computing the scores of the hits
19:        Set $Score(d, q) \leftarrow \sum_{q_c \in C} Score(d, q_c)$  ▷ since $Score$ is decomposable
20:    end for
21:    Sort $A$ wrt $Score(\cdot, q)$
22:    return $A$
23: end if
```

(the answers are sets), while in the second case, we must retain only those documents that appear in all the answers of the cached subqueries.

An alternative method to support *AND* semantics is to use the decomposable hybrid scoring function (Eq. 3) described in Section 3 and consider only those documents whose score is greater or equal than $|t(q)|$. This restriction guarantees that only those documents containing all the query terms of q, will be included in $Ans(q)$, since (at least) 1 unit is added for each query term of q that d contains. In this case, we do not have to change the query evaluation algorithm (Alg. 3) at all.

6 Top-K SCRC, Accuracy/Completeness of the Composed Top-K Answer

Here we elaborate on the case where only the top-K results of each cached query are stored in SCRC. Obviously this increases the utilization of the cache, since a bigger number of queries can be hosted.

Clearly an "identical hit" will return the correct top-K result, however a *"set cover hit"* will not necessarily return the correct top-K result since there may exist a document that belongs to the answers of two or more subqueries but in positions after K, and thus not hosted in the cache. Consequently that document will get less score than what it deserves[2]. Below we analyze this issue and show how we can compute the accuracy of the produced answer.

Let q be the submitted query, let C be *an exact set cover* of q found in the cache, and let B be the documents appearing in the answers of the queries in C, i.e. $B = \cup_{q_c \in C} Ans_{cache}(q_c)$.

For example, consider the Top-K SCRC (here $K = 4$) shown in Figure 4. Suppose that the full answers of the two queries contain 5 documents but the cache stores only the 4 first hits (since $K = 4$); the 5th element, which is not hosted by the cached, is shown in the figure with dotted style at the right side.

Now consider a submitted query q = "$a\ b\ c$". Obviously there is a "set cover hit", $C = \{q_{c_1}, q_{c_2}\}$, and $B = \cup_{q_c \in C} Ans_{cache}(q_c) = \{d_1, d_2, d_3, d_4, d_5\}$. The rising question is whether this is complete and what is the right ordering.

[2] In query independent ranking models we do not have this issue.

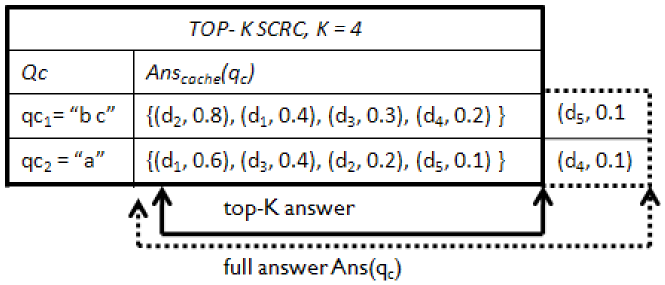


Figure 4: Top-K SCRC Example

In general, if a document b appears in the answers of *all* queries in C, like documents d_1, d_2 and d_3 do, then the final score of b will be the correct one. If b does not appear in any cached answer of the queries in C (i.e. if $b \notin B$), then b will not appear in the final answer. Now consider the case where b appears in *some* of the answers of the queries in C (i.e. its position in the answers of one or more queries in C is $k+1$ or greater). In this case b will get a score *smaller* than what it deserves. In the above example, documents d_4 and d_5 will get a smaller score than what they deserve, since both of them appear in the full answer $Ans(q_c)$, but they are not included in the top-4 cached answers of q_{c_2} and q_{c_1} respectively.

However we can compute an *upper bound* of the score that document b can have by assuming that in those answers where b is absent, b could take at most the score of the last cached element. In the above example, document d_4 could take at most the score of the last element of $Ans_{cache}(q_{c_2})$ which is 0.1 and document d_5 could take at most the score of the last element of $Ans_{cache}(q_{c_1})$ which is 0.2. It follows that for each $b \in B$ we have its *certain score*, denoted by $score_{certain}(b)$ (which is $Sim(b,q)$ as computed by the cache), and an *upper bound for its missed score*, say $score^{up}_{miss}(b)$. If C' ($C' \subset C$) denotes the set of queries which do not have b in their answers, then $score^{up}_{miss}(b) = \sum_{q_c \in C'} \min\{\, score_{certain}(b') \mid b' \in Ans_{cache}(q_c)\}$. If $score^{up}_{miss}(b) = 0$ then this means that b is present in the answers of all subqueries so we know its certain score. Let $score_{up}(b)$ denote the upper bound of the score of b, i.e. $score_{up}(b) = score_{certain}(b) + score^{up}_{miss}(b)$. In general it holds $score_{certain}(b) \leq score(b) \leq score_{up}(b)$.

In our example, the certain, missed and upper bound scores are shown next:

	score		
d_i	certain	up miss	up
d_1	1.0	0	1.0
d_2	1.0	0	1.0
d_3	0.7	0	0.7
d_4	0.2	0.1	0.3
d_5	0.1	0.2	0.3

We can exploit the upper bounds of the missed scores as well as of the scores of the missed documents for ordering the documents and for identifying which proportion of the top-K answer computed by a set cover hit is *guaranteed to be accurate*. We can identify two proportions each described by an integer ranging $[1..K]$, the *exact top-K_{ex} set* and the *correct relative order top-K_{ro} part*, which are the biggest integers that satisfy the following equations:

$$\begin{aligned} Set(top(K_{ex})(Ans_{cache}(q))) &= Set(top(K_{ex})(Ans(q))) \\ top(K_{ro})(Ans_{cache}(q)) &= top(K_{ro})(Ans(q)_{|B}) \end{aligned}$$

where $Ans(q)_{|B}$ denotes the restriction of $Ans(q)$ on B. The first equation ensures that the first K_{ex} are definitely the highest scored elements, while the second ensures that the first K_{ro} elements have the right relative order.

For instance, if

$$\begin{aligned} Ans(q) &= \{d_1, d_2, d_3, d_4\} \\ Ans_{cache}(q) &= \{d_1, d_3, d_2, d_5\} \end{aligned}$$

then $K_{ex} = 3$ and $K_{ro} = 1$.

Let us now see how we can compute the guaranteed values of above metrics, K_{ex} and K_{ro}, at query evaluation time and without of course having access to the full answers.

Let $Found^{up}(B)$ be the maximum $score_{up}(b)$ for all b in B. Let $Missing^{up}$ be the maximum score that a $b \notin B$ can have, which is at most equal to the sum of the scores of the last elements in the answers of the queries C, Formally:

$$\begin{aligned} Missing^{up} &= \sum_{q_c \in C} \min\{\ score_{certain}(b) \mid b \in Ans_{cache}(q_c)\} \\ Found^{up}(B) &= \max\{\ score_{up}(b) \mid b \in B\} \end{aligned}$$

and obviously $Missing^{up} \leq Found^{up}(B)$. In our example, it easily follows that:

$$\begin{aligned} Missing^{up} &= 0.1 + 0.2 = 0.3\text{(the sum of the last scores)} \\ Found^{up}(B) &= 1.0\text{(that of } d_1 \text{ or } d_2\text{)} \end{aligned}$$

From the algorithmic point of view, the values K_{ex} and K_{ro} can be computed in a simple and efficient manner. We order the objects in B in descending order with respect to their *certain* scores. In our example, by ordering B in descending order w.r.t their certain scores (and retaining only the top-4 highest ranked documents) we get:

$$B : \langle (d_1, 1.0), (d_2, 1.0), (d_3, 0.7), (d_4, 0.2) \rangle$$

Subsequently we start from the first and we proceed as long as the certain score of the current element is greater than or equal to $Found^{up}(B \setminus X)$ where X is the set of elements visited so far including the current (so at the beginning X is a singleton holding the element with maximum certain score). The position reached is the K_{ro} that we are looking for. To find K_{ex} we start again from the start of the list and we proceed as long as the certain score of the current element is greater than or equal to $Missing^{up}$. The position reached is the K_{ex} that we are looking for. The proof that the computation of K_{ex} and K_{ro} is correct is given in the appendix. In our example, to find K_{ro} we proceed as long the following inequalities hold:

$$\begin{aligned} score_{cert}(d_1) = 1.0 &\geq Found^{up}(\{d_2, d_3, d_4, d_5\}) = 1.0\checkmark \\ score_{cert}(d_2) = 1.0 &\geq Found^{up}(\{d_3, d_4, d_5\}) = 0.7\checkmark \\ score_{cert}(d_3) = 0.7 &\geq Found^{up}(\{d_4, d_5\}) = 0.3\checkmark \\ score_{cert}(d_4) = 0.2 &\geq Found^{up}(\{d_5\}) = 0.3 \end{aligned}$$

Notice that the first three inequalities hold, so $K_{ro} = 4$. Accordingly, K_{ex} equals 3 as the first three documents have $score_{cert} \geq Missing^{up} = 0.3$.

Let us now compare the answer computed by the Top-K SCRC with that of the main index. If we evaluate the query q from the main index (which has access to the full posting lists of the terms) and then get the top-4 highest ranked documents from the returned answer, then we will get: $Ans(q) : \langle (d_1, 1.0), (d_2, 1.0), (d_3, 0.7), (d_4, 0.3) \rangle$ Notice that in comparison to the answer computed through Top-K SCRC, only the score of d_4 is different. The answer computed by Top-K SCRC is correct since it contains the same documents as $Ans(q)$ and the relative order is correct. One can easily see that the mathematically guaranteed values of K_{ex} and K_{ro}, can be more “strict” than that their actual values.

The above procedure is very fast as no extra index/disc access is required. The ability to compute the above portions allows implementing various policies, e.g. if K_{ex} or K_{ro} are less than a pre-specified number, a policy could be to evaluate the query from scratch.

7 Experimental Evaluation

This section reports extensive and comparative experimental results:
Section 7.1 introduces various *query log metrics* for identifying those characteristics of a query stream that affect the speed up that can be obtained by the SCRC, and Section 7.2 reports experimental results over several query logs of real WSEs.
Section 7.3 reports experimental results about the potential of the *greedy* algorithm in detecting exact set covers.
Section 7.4 introduces various *cache hit metrics* and Section 7.5 reports comparative experimental results (for RC and SCRC) over real query logs.
Section 7.6 reports detailed *performance* results, that concern the achieved *cache hit ratios* (section 7.6.1), *average response times* (section 7.6.2), how these times are analyzed to the various subtasks (section 7.6.3), and the *accuracy* of the top-K answer (section 7.6.4).

7.1 Query Stream Metrics

To characterize a query stream Q_e we introduce four metrics. In the formulae of these metrics we treat Q_e as a *bag* of queries (i.e. duplicates are allowed). Whenever we want to refer to the *set* of distinct queries of Q_e, we will write $Set(Q_e)$.
The *Average Query Length ($AVGQLEN$)* is defined as:

$$AVGQLEN(Q_e) = \frac{\sum_{q \in Q_e} |t(q)|}{|Q_e|}$$

The *Identical Query Ratio (IQR)* is defined as the ratio of the identical queries in the query stream:

$$IQR(Q_e) = 1 - \frac{|Set(Q_e)|}{|Q_e|}$$

To characterize a stream with respect to its density of *partial* exact set covers we use the *Partial Exact Set Cover Density (PESCD)* defined as:

$$PESCD(Q_e) = \frac{|\{q \in Q_e \mid q = PESC(qset), qset \subseteq Set(Q_e) \setminus \{q\}\}|}{|Q_e|}$$

To characterize a stream with respect to the probability of having an *exact* set cover, we introduce the *Set Cover Density (SCD)* which is the proportion of the queries in Q_e whose terms can be covered by the rest queries in Q_e:

$$SCD(Q_e) \quad = \quad \frac{|\{q \in Q_e \mid q = ESC(qset), qset \subseteq Set(Q_e) \setminus \{q\}\}|}{|Q_e|}$$

7.2 Query Stream Metrics: Experimental Results

To measure SCD in real query streams, we computed it over six real query logs of three different WSEs (3 traces from *Excite*[3], 1 trace from *Altavista*[4], and 2 traces from *AllTheWeb*[5]). All queries contained in these logs were firstly

[3]www.excite.com
[4]www.altavista.com
[5]www.alltheweb.com

preprocessed by converting all query terms to lowercase and reordering them in alphabetical order, while stopwords were removed. We removed requests for further result pages (duplicate queries), since such requests are beneficial for WSEs for determining the number of results that the user visited but skew the results in analyzing how the user searched the system. Empty queries, as well as queries containing special characters (like phrase queries and urls) were ignored.

Table 4 reports the statistics for each of these logs. The average SCD value over the above query logs is 37%, the average $PESCD$ value is 27.5% and the average IQR value is 33%.

Table 4: Statistics of real query logs using Algorithm 1

Query Log	Date	$\|Q_e\|$	$SCD(Q_e)$	$PESCD(Q_e)$	$IQR(Q_e)$	$AVGQLEN(Q_e)$
Excite	March 13, 1997	36,923	10%	33.2%	52%	2.03
Excite	Sept. 19, 1997	392,503	43.4%	22.2%	35.5%	2.16
Excite	Dec. 20, 1999	1,125,691	54.1%	18.5%	38.8%	2.25
Excite	May 4, 2001	479,669	41.4%	26.1%	29%	2.16
Altavista	Sep. 9, 2002	1,154,598	39%	35.3%	22.6%	2.88
AllTheWeb	Feb. 6, 2001	462,678	30.7%	35%	26%	2.25
AllTheWeb	May. 28, 2002	764,045	42.6%	22.6%	25.2%	2.11

7.3 Greedy Approximation: Experimental Results

We also used an exhaustive (non-greedy) algorithm for computing the SCD values over these logs, which examines all combinations of the cached sub-queries of the incoming query. Our objective was to verify that the greedy algorithm does not lead to pure retrieval of exact set covers. The results of the exhaustive algorithm are presented in Table 5. By comparing the values of the $SCD(Q_e)$ metric in Table 4 with those of Table 5, we can see that the exhaustive (non-greedy) algorithm offers at most 0.2%-0.6% more exact set covers than the greedy one.

Table 5: Statistics of real query logs using a non-greedy algorithm

Query Log	Date	$\|Q_e\|$	$SCD(Q_e)$
Excite	March 13, 1997	36,923	10.2%
Excite	Sept. 19, 1997	392,503	44%
Excite	Dec. 20, 1999	1,125,691	54.5%
Excite	May 4, 2001	479,669	41.8%
Altavista	Sep. 9, 2002	1,154,598	39.4%
AllTheWeb	Feb. 6, 2001	462,678	31%
AllTheWeb	May. 28, 2002	764,045	42.8%

7.4 Cache Hit Metrics

The value of the SCD metric in a query stream is a good indicator for revealing how many queries can be covered exactly by the rest queries and indicates the maximum number of set cover hits that a SCRC can achieve. However, it cannot guarantee that all of these queries will be answered though set cover hits. For instance, if set cover queries are very popular then they will be cached (assuming that the cache stores the most frequent queries) and they will be answered by the cache as *identical hits*, since we always firstly check for identical cached queries. Another reason that these queries may not be answered through the cache is due to memory limitations. If all the *lower* queries of a set cover query are not cached, then it cannot be answered exclusively through the cache.

Here we refine the metrics defined earlier in order to count the number of identical, partial exact set covers or *set cover hits* that can be found over the *cached queries* Q_c. Specifically:

- The metric $IQR(Q_e, Q_c)$ counts the number of the identical hits that can be found over the cached queries Q_c and is defined as:

$$IQR(Q_e, Q_c) = \frac{|\{q' \equiv q \mid q \in Q_e, q' \in Q_c\}|}{|Q_e|}$$

- The metric $PESCD(Q_e, Q_c)$ counts the number of the partial exact set covers that can be found over the cached queries Q_c and is defined as:

$$PESCD(Q_e, Q_c) = \frac{|\{q \in Q_e \mid q = PESC(qset), qset \subseteq Q_c \setminus \{q\}\}|}{|Q_e|}$$

- The metric $SCD(Q_e, Q_c)$ counts the number of the exact set covers that can be found over the cached queries Q_c and is defined as:

$$SCD(Q_e, Q_c) = \frac{|\{q \in Q_e \mid q = ESC(qset), qset \subseteq Q_c \setminus \{q\}\}|}{|Q_e|}$$

Notice that the nominator of SCD contains $qset \subseteq Q_c \setminus \{q\}$, consequently the identical queries of q are not taken into account (they are considered only at the computation of IQR).

The cache hit rate of the RC is equal to the fraction of the identical hits, while the cache hit rate of the SCRC is equal to the sum of the fraction of the identical hits and that of the exact set cover hits. Hence, for result caches of capacity $|Q_c|$ the cache hit rate of the RC is equal to the $IQR(Q_e, Q_c)$ value, while the cache hit rate of the SCRC is equal to the sum of the $IQR(Q_e, Q_c)$ and the $SCD(Q_e, Q_c)$ values.

Since, the incoming query can be answered by the cache either through an identical hit or by an exact set cover hit or by a partial exact set cover, the sum of the values of the cache metrics ($IQR(Q_e, Q_c)$, $PESCD(Q_e, Q_c)$, $SCD(Q_e, Q_c)$) for a certain cache size must always be ≤ 1.

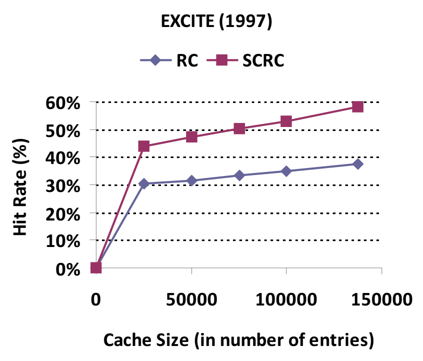
7.5 Cache Hit Metrics: Experimental Results

In order to evaluate the performance of the RC and the SCRC we computed their hit rate by partitioning each query log into two parts of equal size: a *training set* and a *test set*, as in [19]. We use the training set to extract the most *frequent* queries and fill the caches and we use the test set to measure the cache hit rate. The contents of both caches (RC, SCRC) are *static* and do not change during the submission of the queries contained in the test set.

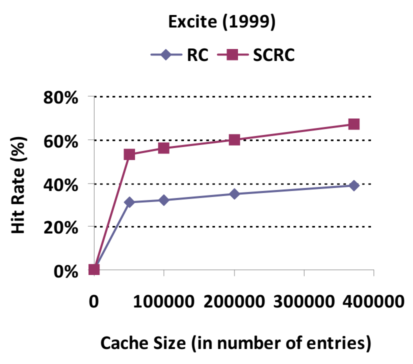
Next, we report the cache hit rate of the RC and the SCRC for various cache sizes over the query logs. We also report for each cache size the fraction of the identical hits, the fraction of the exact set cover hits and the fraction of the partial exact set covers. Note that we do not count the fraction of the partial exact set covers in the cache hit rate of the SCRC.

[Log 1. Excite (1997)]
Figure 5(a) shows the cache hit rate of the RC and the SCRC over the Excite (1997) query log. In this stream, we observe that the SCRC attains 3.9% - 20.4% higher hit rates than the RC. Table 6 reports in detail the fraction of the identical hits, the exact set covers and the partial exact set covers that were obtained for each cache size. We observe that as the cache size increases:

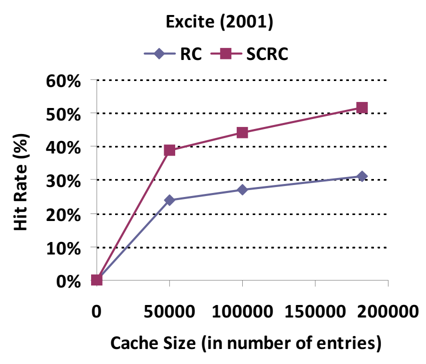
- the fraction of the identical hits also increases
- the fraction of the exact set cover hits also increases almost at the same rate as the identical hits
- the fraction of the partial exact set covers decreases but at a lower rate



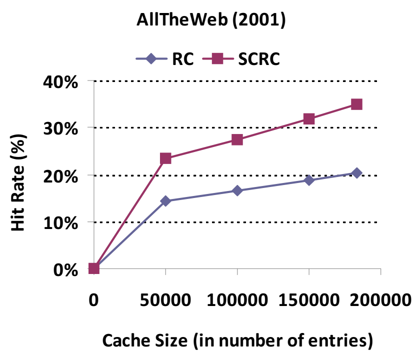
(a) *Excite (1997) log*



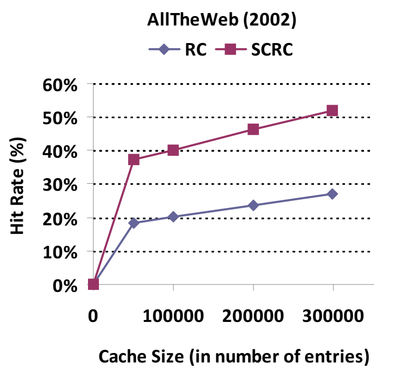
(b) *Excite (1999) log*



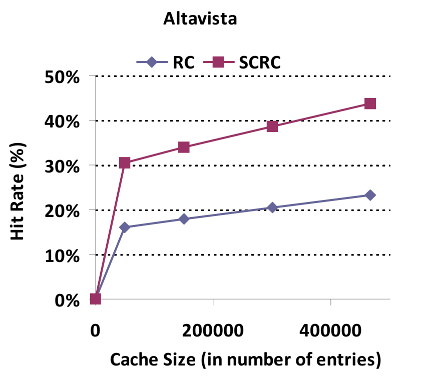
(c) *Excite (2001) log*



(d) *AllTheWeb (2001) log*



(e) *AllTheWeb (2002) log*



(f) *Altavista log*

Figure 5: Hit Rate of the RC and the SCRC as a function of the Cache Size

[Log 2. Excite (1999)]
Figure 5(b) shows the cache hit rate of the RC and the SCRC over the Excite (1999) query log. We see that the SCRC attains 3.7% - 28% higher hit rates than the RC, which is higher than the obtained hit rate in the Excite (1997) log. Accordingly, Table 7 reports in detail the fraction of the identical hits, the exact set covers and the partial exact set covers that were obtained for each cache size. The fraction of the exact set cover hits are higher than in the Excite (1997) log but the fraction of the partial exact set covers is almost the same.

[Log 3. Excite (2001)]
Figure 5(c) shows the cache hit rate of the RC and the SCRC over the Excite (2001) query log. In this log, we observe that the SCRC attains 2.6% - 20.5% higher hit rates than the RC.

Table 8 shows the fraction of the identical, exact set cover, and partial exact set cover hits for each cache size. The rate at which the identical and the exact set cover hits increase and the partial exact set covers decrease is similar to those of the previous logs.

[Log 4. AllTheWeb (2001)]
Now, we study the hit rate of the RC and the SCRC of the logs of the AllTheWeb WSE. Figure 5(d) shows the cache hit rate of the RC and the SCRC over the AllTheWeb (2001) query log. We see that the SCRC attains 1.6% - 14.6% higher hit rates than the RC.

Table 9 shows the fraction of the identical, exact set cover, and partial exact set cover hits for each cache size. In this log, we observe that the hit rate of the SCRC is lower than the one in the previously examined logs of the *Excite* WSE. On the other hand, the fraction of the partial exact set covers is higher than the one in the previous logs and remains almost stable for all cache sizes.

[Log 5. AllTheWeb (2002)]

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	13.4%	3.9%	26.1%
2,500	18.4%	6.7%	30.3%
5,000	22.4%	9.3%	32%
10,000	26.3%	12%	31.7%
25,000	30.4%	13.6%	30.4%
50,000	31.6%	15.8%	29.6%
75,000	33.2%	17.1%	28.6%
100,000	34.9%	18.1%	27.4%
137,653	37.6%	20.4%	25.3%

Table 6: Fraction of identical hits, exact set cover hits and partial exact set covers over the Excite (1997) query log

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	10.2%	3.7%	27.8%
5,000	18%	9.9%	36.4%
10,000	22.1%	13.8%	36.8%
25,000	27.3%	19.4%	33.4%
50,000	31%	22%	30%
100,000	32%	24%	28.8%
200,000	34.8%	25.2%	26.6%
371,756	39%	28%	22.2%

Table 7: Fraction of identical hits, exact set cover hits and partial exact set covers over the Excite (1999) query log

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	10.3%	2.6%	23.4%
5,000	17.3%	7.8%	34%
10,000	20.3%	10.5%	34.9%
25,000	23.6%	13.7%	34.4%
50,000	24%	15%	33.8%
100,000	27.1%	17.1%	32.4%
181,954	31%	20.5%	29.4%

Table 8: Fraction of identical hits, exact set cover hits and partial exact set covers over the Excite (2001) query log

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	4.5%	1.6%	20%
5,000	8.4%	3.9%	28.6%
10,000	10.6%	5.7%	32.6%
25,000	13.1%	8%	35.5%
50,000	14.3%	9.1%	36.3%
100,000	16.6%	10.8%	36.9%
150,000	18.8%	13%	36.7%
183,000	20.4%	14.6%	36.4%

Table 9: Fraction of identical hits, exact set cover hits and partial exact set covers over the AllTheWeb (2001) query log

Figure 5(e) shows the cache hit rate of the RC and the SCRC over the AllTheWeb (2002) query log. We see that the SCRC attains 3% - 25% higher hit rates than the RC.

Table 10 shows the fraction of the identical, exact set cover, and partial exact set cover hits for each cache size.

[Log 6. AltaVista]

Finally, Figure 5(f) shows the cache hit rate of the RC and the SCRC over the Altavista query log. We see that the SCRC attains 1.5% - 20.5% higher hit rates than the RC.

Table 11 shows the fraction of the identical, exact set cover, and partial exact set cover hits for each cache size. It is worth stating that in the particular query log the fraction of the partial exact set covers is the highest among all the other logs.

Summary. We studied and analyzed logs from 3 real WSEs. Our results indicate that over all query logs there is a significant fraction of queries that can be covered (either totally or partially) by the rest submitted queries. Using trace driven simulations for various cache sizes we showed that the SCRC outperforms the RC in all cases even when the cache size is set to a few cached entries. More precisely, we showed a SCRC can deliver up to 28% higher hit rates than a RC. Moreover, we showed that as the cache size increases, the fraction of the identical hits and the exact set cover hits also increases (almost at the same rate), while the fraction of the partial set covers is slightly decreased in most of the cases. For a given cache size, we showed that about half of the queries can always be covered either totally (exact set cover) or partially (partial exact set cover). This implies that even if there is not an exact set cover,

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	5.4%	3%	23.1%
5,000	10.4%	8.7%	34.1%
10,000	13%	12.2%	35.3%
25,000	16.5%	16.5%	34.6%
50,000	18.4%	18.8%	33.5%
100,000	20.2%	20%	32.2%
200,000	23.6%	22.8%	29.8%
297,842	27.1%	24.9%	26.9%

Table 10: Fraction of identical hits, exact set cover hits and partial exact set covers over the AllTheWeb (2002) query log

Cache Size	IQR $(Q_e, \cdot)$	SCD $(Q_e, \cdot)$	PESCD $(Q_e, \cdot)$
1,000	4.5%	1.5%	19.8%
5,000	8.5%	5.4%	32%
10,000	10.6%	8%	35.2%
25,000	13.6%	11.6%	37.7%
50,000	16.1%	14.4%	38.1%
150,000	17.9%	16.1%	38.3%
300,000	20.5%	18.2%	38%
466,033	23.2%	20.5%	37.2%

Table 11: Fraction of identical hits, exact set cover hits and partial exact set covers over the Altavista query log

there is a significant fraction of queries that constitute partial exact set covers and can potentially have a great impact on the performance of a WSE by reducing the costs of the query evaluation process.

7.6 Performance Results

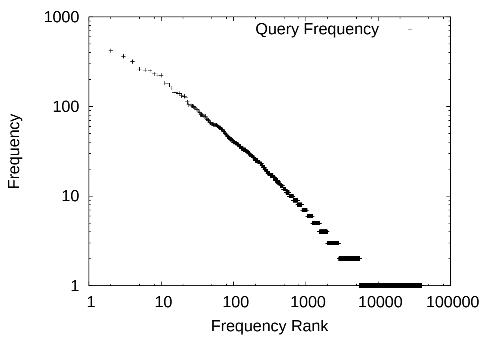
In this section, we describe and report our results from the experimental evaluation over the Mitos WSE [21].
Hardware and Software Environment. Mitos WSE has two releases: one based on a classical Inverted File index (as in Terrier), and one based on an index represented in an Object-Relational DBMS. Preliminary experiments showed that the performance gain achieved either by a SCRC or a RC is roughly the same in both releases, so we report results using the DBMS-based release. More specifically, we verified experimentally (for various cache sizes) that when a caching mechanism (either RC or the SCRC) was employed by the Mitos WSE, the reduction in the average query response time of the WSE remained the same in both the DBMS-based release and the Inverted File release. All experiments were carried out by running a single process on a desktop PC with a Pentium IV 3.4 GHz processor, 2 GB main memory on top of Linux distribution.
Performance Measures. To make the results comparable all implementations take as input a Max Cache Memory parameter M. Since the exact costs for the evaluation of a query rely on the internal design and hardware environment of the WSE, we used two measures to estimate the efficiency of the caching mechanisms:

- *Cache Hit Ratio*: the ratio of the queries that are answered by the cache. This measure has the advantage of being independent of both the software and the hardware environment.
- *Average Query Execution Time:* the average response time of the Mitos WSE for evaluating a query.

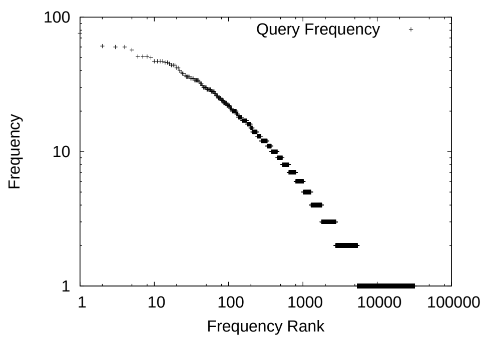
In all experiments presented in this section we used the Okapi BM25 weighting scheme. Results for other retrieval models using *decomposable* scoring functions (i.e VSM) are expected to be almost the same for the reasons we justified in Section 5.3.
Data Set. Our document collection (data set) is a set of web pages crawled by the Mitos web crawler. The index contains 240,108 distinct terms. The total size of the pages is 5.6 GB, yielding a DBMS-based index of approximately 170 MB without positional information.
Query Sets. We used two real query logs, one from *Altavista* (Sept. 9, 2002) and another from *Excite* (Sept. 19, 1997). Since, we are not affiliated with any major WSE, we considered only those queries of which all terms appear in the vocabulary of our index.

- *AltaVista Set*: From the original query set, 70,710 queries remained and 56.4% of these were unique.
- *Excite Set*: From the original query set, 50,000 queries remained and 63.6% of these queries were unique.

Table 12 shows the values of the metrics for the query sets. The characteristics of these query sets are very similar to the characteristics of the real query logs (Table 4). Figures 6 (a) and (b) show the distributions of the queries. The



(a) *Altavista* query set



(b) *Excite* query set

Figure 6: The distribution of the queries

x-axis represents the normalized frequency rank of the query, that is, the most frequent query appears closest to the y-axis. The y-axis is the normalized frequency for a given query. Both axes are on a log scale.

Table 12: Query Set Metrics

Query Set	Metrics				
	$\|Q_e\|$	$IQR(Q_e)$	$SCD(Q_e)$	$PESCD(Q_e)$	$AVGQLEN(Q_e)$
AltaVista	70,710	43.6%	35.3%	20.1%	2.0
Excite	50,000	36.4%	38.2%	38%	2.2

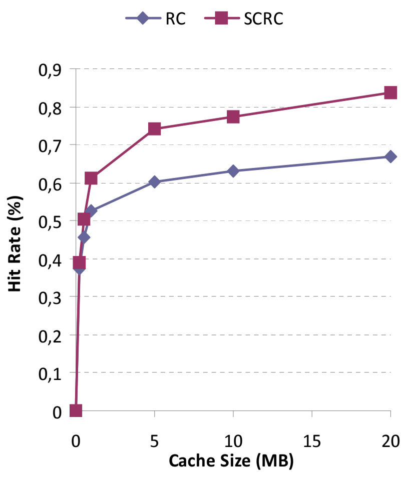
Cache Filling. Regarding the initialization of the caches, we follow the same strategy as before. We split each query trace into 2 equal size parts: a *training set* and a *test set*. The training set is used for filling the caches with the most frequent queries and their top-100 answers. The test set is used for submitting these queries to the Mitos WSE and evaluating the performance of the RC and the SCRC. In both caches, each element in $Ans_{cache}(q_c)$ consists of pairs of the form $(d_i, Score(d_i, q_c))$; hence, both caches retain the same content.
The first kind of experiments (described in Section 7.6.1) aims to assess the hit rate of each caching mechanism and the second kind of experiments (described in Section 7.6.2) aims to measure the average query execution time when the WSE employs either the SCRC or the RC.

7.6.1 Cache Hit Ratio

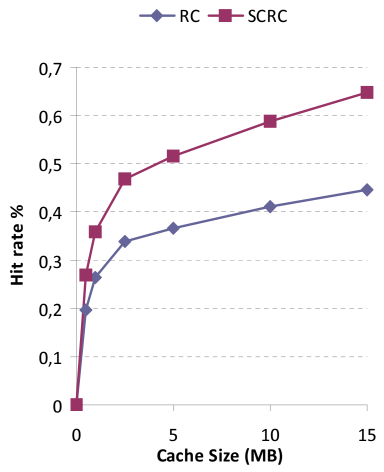
Figures 7 (a) and (b) show the cache hit ratio of the RC and the SCRC over the *Altavista* and the *Excite* query set respectively for various cache sizes. The hit ratio increases as the cache capacity increases for both the RC and the SCRC. Over both query sets, the SCRC achieves higher hit rates, since always a significant fraction of the submitted queries are answered by the SCRC as *exact set cover hits*. More precisely, over the *Altavista* query set the SCRC answers up to 16.8% more queries than the RC and over the *Excite* query set the SCRC achieves up to 20% higher hit rates than the RC. Recall that we do not count the fraction of the partial exact set covers in the cache hit rate of the SCRC.

7.6.2 Average Response Times

Now we report the average query response times of the Mitos WSE for various cache sizes. We conducted this second kind of experiments in order to verify that the higher hit rate of the SCRC also results in lower query response times



(a) *Altavista* query set



(b) *Excite* query set

Figure 7: Top-K SCRC Hit Rate Vs Top-K RC Hit Rate

of the WSE. Recall that both the RC and the SCRC are implemented as hash-based data structures. Regarding the query evaluation process when Mitos employs the RC, we use Algorithm 3 presented in Section 5.3. In both cases, when a cache (either the RC or the SCRC) reports a miss, the query is fully evaluated from the main index.
Altavista query set: Figure 8 illustrates the average query response time of the Mitos WSE when employing the RC and the SCRC over the *Altavista* set. The first column depicts the average query evaluation time by using the answer of the index (no cache). The second and the third column show the average response time of Mitos WSE when using the RC and the SCRC respectively.

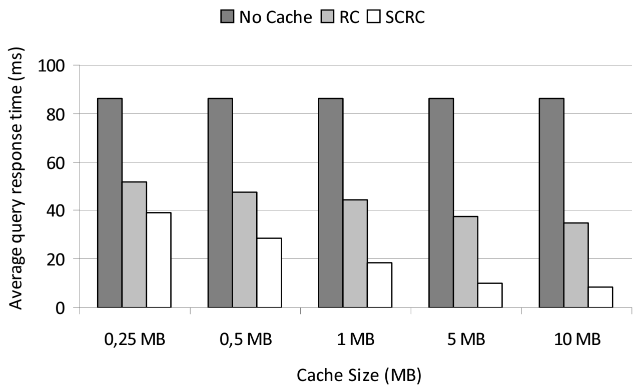


Figure 8: Average query response time over the *Altavista* query set (in ms)

Firstly, we observe that the use of any caching mechanism (either the RC or the SCRC) always reduces the average query response time of the WSE. Moreover, we observe that the SCRC outperforms the RC in all cases, even when the cache size is very small. As the cache size increases the speedup obtained by the SCRC compared to the RC also increases significantly. When the cache size is medium (i.e *M = 1 MB*), SCRC is at least 2 times faster than the RC. When the cache size is large (i.e *M = 10 MB*), SCRC is 4 times faster than the RC.

To understand the extra speedup that the SCRC attains, we have to measure both (a) the fraction of the identical hits since they contribute to the SCRC speedup and (b) the exact computational costs for answering the submitted

Cache Size	IQR (Q_e, $\cdot$)	SCD (Q_e, $\cdot$)	PESCD (Q_e, $\cdot$)
0.25 MB	37.5%	2.5%	23.3%
0.5 MB	45.7%	4.6%	20.4%
1 MB	52.5%	8.8%	22.6%
5 MB	60.2%	14%	17.7%
10 MB	63%	14.4%	15.9%
20 MB	67%	16.8%	12.1%

Table 13: Fraction of identical hits, exact set cover hits and partial exact set covers over the *Altavista* query set

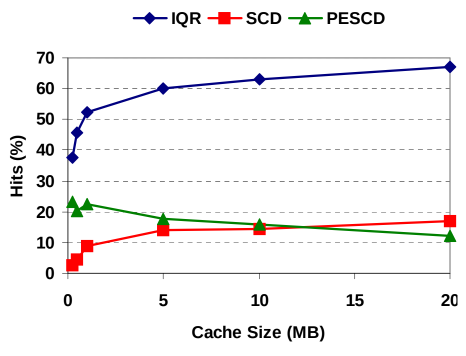
Cache Size	IQR (Q_e, $\cdot$)	SCD (Q_e, $\cdot$)	PESCD (Q_e, $\cdot$)
0.5 MB	19.8%	7.2%	32.4%
1 MB	26.3%	9.6%	30%
2.5 MB	33.8%	13%	25.4%
5 MB	36.6%	15%	24%
10 MB	41.8%	17%	22.5%
15 MB	44.7%	20%	21.1%

Table 14: Fraction of identical hits, exact set cover hits and partial exact set covers over the *Excite* query set

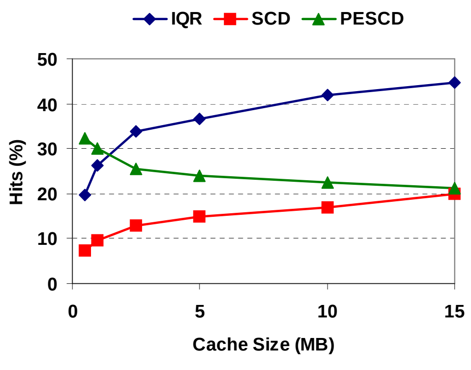
query through an identical hit, an exact set cover hit, a partial exact set cover and through the main index of the Mitos WSE. Notice that the queries that are answered by the SCRC either or exact set cover hits or partial exact set covers will be definitely RC cache misses in the case of the RC (and hence they will be evaluated from the scratch by the main index). Section 7.6.3 reports these costs in detail.

Table 13 shows the fraction of the identical hits, the exact set cover hits and the partial exact set covers for each cache size.

The same results are illustrated as plots in Figure 9(a). We observe that as the cache size increases: the fraction of the identical hits and the exact set cover hits increases and the fraction of the partial exact set covers decreases. Moreover, we observe that as the number of the exact set cover hits increases, the average query execution time decreases, despite the fact that the fraction of the partial exact set covers also decreases. This verifies that the use of the exact set covers for the query evaluation process is much more beneficial than the use of the partial exact set covers.



(a) Altavista query set



(b) Excite query set

Figure 9: Fraction of identical hits, exact set cover hits and partial exact set covers

Excite query set: Below we report the results of analogous measurements over the query log of Excite.

Figure 10 illustrates the average query response time of the Mitos WSE when employing the RC and the SCRC over the *Excite* query set. Again the SCRC is faster than the RC in all cases, even when the cache size is very small.

As the cache size increases the speedup obtained by the SCRC compared to the RC also increases. When the cache size is medium (i.e *M = 5 MB*), SCRC is 2 times faster than the RC. When the cache size is larger (i.e *M = 15 MB*), SCRC is 3 times faster than the RC.

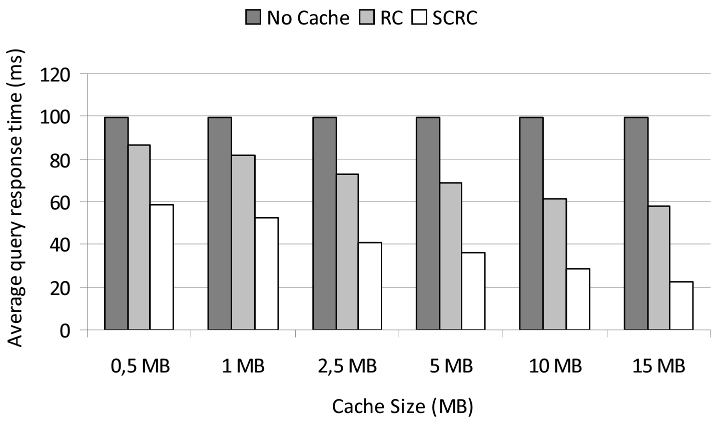


Figure 10: Average query response time over the *Excite* query set (in ms)

Table 14 shows the fraction of the identical hits, the exact set cover hits and the partial exact set covers for each cache size. The same results are illustrated as plots in Figure 9(b).

7.6.3 Analysis of Time Spent

In the previous section, we mentioned that the use of the exact set covers for the query evaluation process is much more efficient than the use of the partial exact set covers. This can be shown more clearly in Table 15, where we report the average query execution time (in milliseconds) to answer a query of the *Altavista* query set through:

- the index of the WSE
- an identical cached query
- an exact set cover (ESC)
- a partial exact set cover (PESC)

The same results (of Table 15) are illustrated as plots in Figure 11(a) where the time axis is in log scale. Notice that the time to answer a query through the index of the WSE or through an identical cached query is the same when the WSE employs either the RC or the SCRC. We can see that identical and exact set cover hits are two to four orders of magnitude faster than PESC and index-based evaluation. Specifically, the time to answer a query q through an identical cached query $q' \equiv q$ is almost negligible (150 microseconds). The average time for obtaining the answer of q through an exact set cover (ESC) is less than 1.5 ms, which is almost two orders of magnitude faster than answering the query through the index (80 ms). We also observe that as the cache size increases, the average time to derive an answer through the use of partial exact set covers ($PESC$) decreases but their benefits are higher. This can be explained by the fact that when the cache increases the size of these partial exact set covers also increase, and this results in acquiring shorter queries (as remainders) from the main index.

The corresponding results for the case of the Excite query log are shown in Table 16 and Figure 11(b). As before, we observe that returning the answer of an identical cached query requires just a few microseconds and returning the answer through an exact set cover hit is hardly 1 ms. Moreover, when the cache size is large, returning the answer through a partial exact set cover requires 50% less time than evaluating it from the main index.

[Analyzing the Cost of a Set Cover Hit]

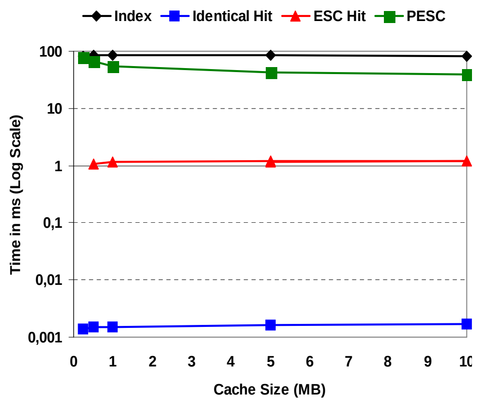
Table 17 shows in detail the average execution time (in microseconds) spent for answering a query through a set cover hit in the Altavista query log. The same results are shown in Figure 12(a) as a bar chart. Specifically, we report the execution time which is spent in order to:

Cache Size	Index	Identical hit	ESC hit	PESC
0.25 MB	82.0	0.00139	1.07	79.2
0.5 MB	85.3	0.00146	1.14	66.7
1 MB	86.7	0.00151	1.2	53.6
5 MB	84.4	0.00162	1.19	42.6
10 MB	81.0	0.00168	1.14	39.4

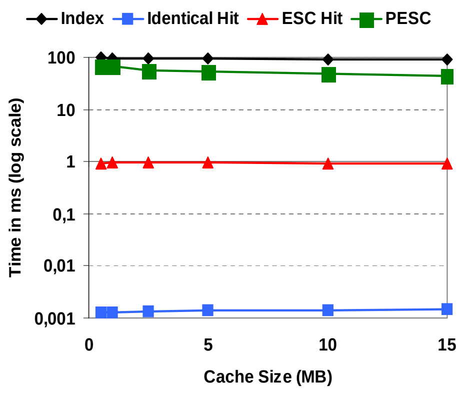
Table 15: Average execution times (in ms) to answer a query through: the index, an identical cached query, an exact set cover hit (ESC) or a partial exact set cover (PESC) over the *Altavista* query set

Cache Size	Index	Identical hit	ESC hit	PESC
0.5 MB	97.7	0.00125	0.909	69
1 MB	96.9	0.00130	0.972	69
2.5 MB	95.3	0.00132	0.958	54.9
5 MB	93.5	0.00137	0.956	52.3
10 MB	91.8	0.00141	0.933	48.6
15 MB	92.0	0.00145	0.914	43.6

Table 16: Average execution times (in ms) to answer a query through: the index, an identical cached query, an exact set cover hit (ESC) or a partial exact set cover (PESC) over the *Excite* query set



(a) Altavista query set



(b) Excite query set

Figure 11: Average execution times (in ms) to answer a query through: the index, an identical cached query, an exact set cover hit (ESC) or a partial exact set cover (PESC)

- produce the lower queries of the incoming query and check in the cache if they exist (Lower Tm),
- execute the greedy algorithm for returning a set cover (Greedy Tm),
- aggregate the answers of the cached queries that constitute the set cover (Aggr. Tm),
- estimate the minimum accuracy of the composed top-K answer (Acc. Tm)

We observe that most of the time is spent on aggregating the answers of the cached subqueries that cover the incoming query (90% of the total time). The average time for generating the sub-queries of each query (using the *hash-based* approach presented in Section 5.1) and examining if the cache contains them is approximately 45 microseconds. The time that the greedy algorithm requires is approximately 16 microseconds. The average time to aggregate the answers of the cached queries is almost 1 ms (1000 microseconds). Finally, the time spent on estimating the minimum accuracy of the composed top-K answer (K_{ex} and K_{ro} values) is 45 microseconds.

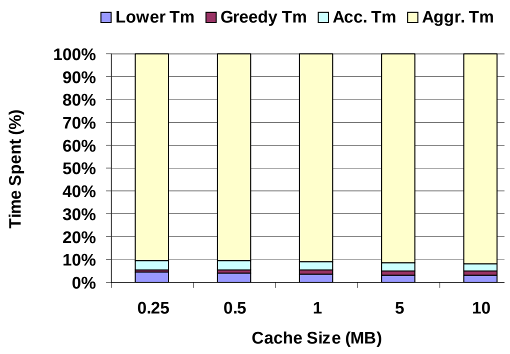
The corresponding results for the Excite query log are shown in Table 18 and Figure 12(b). The results are very similar as those in the *Altavista* set. Most of the time for exploiting a set cover is consumed over aggregating the answers of the cached subqueries of the incoming query.

Cache Size	Lower Tm	Greedy Tm	Aggr. Tm	Acc. Tm
0.25 MB	47.4	9.3	979	44
0.5 MB	49.0	13.2	1038.0	46.8
1 MB	46.3	16.9	1096.4	46.05
5 MB	40.8	20.1	1096.6	40.1
10 MB	39.0	17.2	1052.9	39.4

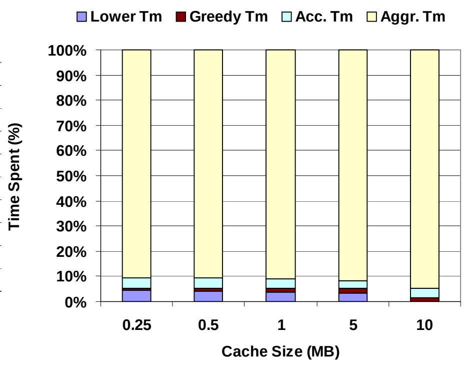
Table 17: Detailed execution times (in microseconds) for deriving the answer of a query through a set cover hit over the *Altavista* query set

Cache Size	Lower Tm	Greedy Tm	Aggr. Tm	Acc. Tm
0.5 MB	64	14	764	67
1 MB	76	16.7	828	52
2.5 MB	58	20.5	828.5	51
5 MB	60	19.5	827	50.4
10 MB	44.1	18.2	824	47
15 MB	37.7	16.9	815	45

Table 18: Detailed execution times (in microseconds) for deriving the answer of a query through a set cover hit over the *Excite* query set



(a) *Altavista query set*



(b) *Excite query set*

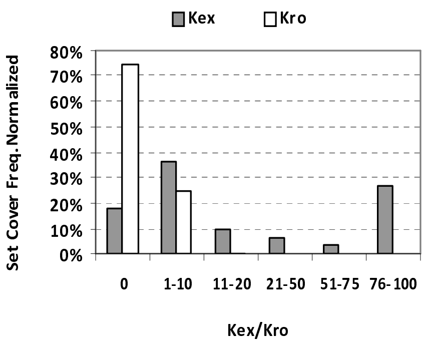
Figure 12: Detailed execution times (in microseconds) for deriving the answer of a query through a set cover hit

7.6.4 Accuracy of the Composed Top-K Answer

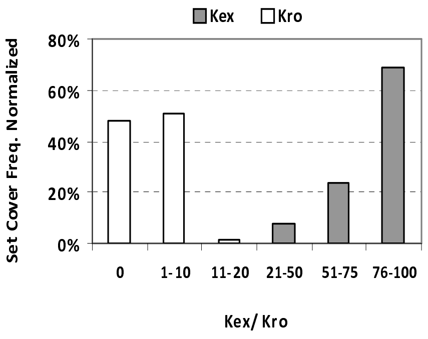
Here we report results regarding the accuracy of the composed top-K answers, assuming $K = 100$, for those queries that were answered through *set cover hits*. We report results over the *Altavista* query set when the cache size M is set to 10 MB; results for the *Excite* query set and other values of M are very similar. Firstly, we estimated the mathematically guaranteed minimum accuracy of these answers using the approach presented in Section 6. The distribution of the K_{ex} and K_{ro} values is illustrated in Figure 13 (a). The x-axis ranges $[0..K]$, i.e. $[0..100]$, and the y-axis ranges $[0\%..100\%]$. The y-value of a x-value is the percentage of the queries that are answered through set cover hits and their accuracy (K_{ex}, K_{ro}) is equal to x value. For instance, the last grey bar of Figure 13 (a) (which corresponds to K_{ex}) illustrates that for the 28% of these queries it is mathematically guaranteed that their composed answer will contain at least the exact top-75 documents. The average K_{ex} value is 43 and the average K_{ro} value is 3.

Next, using the same configurations (same M and K values) as before, we measured the actual accuracy of the composed top-K answer using the main index. Specifically for each query which was answered through a *set cover hit* from the SCRC, we also evaluated it from the main index (by considering the first top-K results) and we compared the returned answers of the Top-K SCRC and that of the main index. Figure 13 (b) shows the accuracy of the composed top-K answer.

The most important observation from this figure is the high level of accuracy that we obtain when evaluating queries through *set cover* hits. More precisely, the average K_{ex} value is two times bigger in comparison to the previous approach (83 vs. 43). This means that in practice the answer through a set cover hit contains 83% of the



(a)Mathematically guaranteed accuracy



(b)Index-based accuracy

Figure 13: Estimating the accuracy of the composed top-K answer for *set cover* hits

top-scored documents. Moreover, for almost all of these queries (99.7%), their answers contain at least the top-20 documents. Furthermore, for 70% of these queries, their answers contain at least the top-75 matching documents. These results are important since previous work [14, 23] on user's web search behavior has shown that users are interested and view only the first pages (1-3) of results and ignore the rest. Regarding the K_{ro} values the obtained results are close to those of the previous approach, but a smaller fraction of these answers have zero K_{ro} values.

8 Concluding Remarks

This paper introduced SCRC, a novel scheme for results caching. The key point is that even if the submitted query does not exist in the cache, nor its constituent terms, it can be answered from the cache if it can be expressed as a disjoint union of the terms of the cached queries. Another important point is that if the ranking of the IRS is based on an additively decomposable scoring function (and we have shown that several classical best-match retrieval models, as well as hybrid retrieval models, rely on such scoring functions), then the SCRC-based query evaluation method derives the right ranking.

The experimental results over real world query traces have demonstrated the superiority of the SCRC over the RC in best-match retrieval models (i.e Okapi BM25, VSM), specifically it is 2-3 times faster than a plain RC (assuming the same memory requirements).

Although the SCRC cache miss time is higher than that of RC, since in SCRC if there is not an identical cached query we have to produce the lower queries of the incoming query and execute the greedy algorithm in order to find an exact set cover hit, we showed that the benefits of the SCRC are not compensated by its higher cache miss time. In a Top-K SCRC, the only price to pay, in comparison to RC, is that if a query is answered by an exact set cover then the ranking of the composed answer is not necessarily the same with that of a answer computed from scratch. However, we have shown how to measure the quality of the composed answer and our experiments showed that in most cases, the first page of results of the composed answer is exact, i.e. it contains the same hits. In bigger indexes the benefits of the SCRC are expected to be even higher, since the cost of query processing is dominated by the cost of traversing the inverted lists, which grow linearly with the collection size.

A challenging direction for further research is to investigate a similar in spirit approach for *reach* answers, i.e. for cases where the query answer is not a simple ranked list of hits, but it also contains groupings of the hits according to various static or dynamically mined metadata, e.g. as in [20, 9]. Since the computation of such answers is computationally more expensive, a caching mechanism that exploits the cached answers of past subseries, would be very beneficial.

Another interesting topic is to consider the same problem but with more complex query languages and/or additional information. For instance, for cases where extra information about the terms is available, e.g. synonymy relationships, one could investigate Rough Sets approximations [31]. For cases where terms are organized hierarchically and boolean expressions are allowed, then approximations like those in [27] could be investigated.

Acknowledgements

We would like to thank Jim Jansen for providing us the query logs of the Excite, the Altavista and AllTheWeb.com WSE. Many thanks also to V. Christophides and E. Markatos for their fruitful comments and suggestions on earlier stages of this work, to Panagiotis Papadakos and Christina Lantzaki for proof reading the manuscript, as well as to the anonymous reviewers for their constructive comments.

References

[1] Baeza-Yate, R., Junqueira, F., Plachouras, V., Witschel, H.: Admission policies for caches of search engine results. In: String Processing and Information Retrieval, pp. 74–85. Springer (2007)

[2] Baeza-Yates, R., Gionis, A., Junqueira, F.P., Murdock, V., Plachouras, V., Silvestri, F.: Design trade-offs for search engine caching. ACM Trans. Web **2**(4), 1–28 (2008). DOI http://doi.acm.org/10.1145/1409220.1409223

[3] Baeza-Yates, R., Junqueira, F., Plachouras, V., Witschel, H.: Admission Policies for Caches of Search Engine Results. LECTURE NOTES IN COMPUTER SCIENCE **4726**, 74 (2007)

[4] Baeza-Yates, R., Saint-Jean, F.: A three level search engine index based in query log distribution. In: String Processing and Information Retrieval, pp. 56–65. Springer (2003)

[5] Baeza-Yates, R.A., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: SIGIR, pp. 183–190 (2007)

[6] Baeza-Yates, R.A., Saint-Jean, F.: A three level search engine index based in query log distribution. In: SPIRE, pp. 56–65 (2003)

[7] Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.: Efficient query evaluation using a two-level retrieval process. In: CIKM '03: Procs of the 12th intern. conf. on Information and knowledge management, pp. 426–434. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/956863.956944

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)

[9] Fafalios, P., Kitsos, I., Marketakis, Y., Baldassarre, C., Salampasis, M., Tzitzikas, Y.: Web searching with entity mining at query time. Multidisciplinary Information Retrieval pp. 73–88 (2012)

[10] 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 Trans. Inf. Syst. **24**(1), 51–78 (2006)

[11] Jansen, B., Pooch, U.: A review of web searching studies and a framework for future research. Journal of the American Society for Information Science and Technology **52**(3), 235–246 (2000)

[12] Jansen, B., Spink, A.: An analysis of web searching by european alltheweb. com users. Information Processing & Management **41**(2), 361–381 (2005)

[13] Jansen, B., Spink, A., Saracevic, T.: Real life, real users, and real needs: a study and analysis of user queries on the web. Information processing & management **36**(2), 207–227 (2000)

[14] Jansen, B.J., Spink, A., Saracevic, T.: Real life, real users, and real needs: a study and analysis of user queries on the web. Inf. Process. Manage. **36**(2), 207–227 (2000). DOI http://dx.doi.org/10.1016/S0306-4573(99)00056-4

[15] Karp, R.: Reducibility among combinatorial problems. Complexity of computer computations **43**, 85–103 (1972)

[16] Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: Procs of the 12th intern. conf. on World Wide Web, pp. 19–28. ACM New York, NY, USA (2003)

[17] Long, X., Suel, T.: Three-Level Caching for Efficient Query Processing in Large Web Search Engines. World Wide Web **9**(4), 369–395 (2006)

[18] Ma, H., Wang, B.: User-aware caching and prefetching query results in web search engines. In: Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval, pp. 1163–1164. ACM (2012)

[19] Markatos, E.: On caching search engine query results. Computer Communications **24**(2), 137–143 (2001)

[20] Papadakos, P., Armenatzoglou, N., Kopidaki, S., Tzitzikas, Y.: On exploiting static and dynamically mined metadata for exploratory web searching. Knowledge and information systems **30**(3), 493–525 (2012)

[21] Papadakos, P., Theoharis, Y., Marketakis, Y., Armenatzoglou, N., Tzitzikas, Y.: Mitos: Design and evaluation of a dbms-based web search engine. In: Informatics, 2008. PCI'08. Panhellenic Conference on, pp. 49–53. IEEE (2008)

[22] Saraiva, P.C., de Moura, E.S., Fonseca, R.C., Jr., W.M., Ribeiro-Neto, B.A., Ziviani, N.: Rank-preserving two-level caching for scalable search engines. In: SIGIR, pp. 51–58 (2001)

[23] Silverstein, C., Marais, H., Henzinger, M., Moricz, M.: Analysis of a very large web search engine query log. SIGIR Forum **33**(1), 6–12 (1999). DOI http://doi.acm.org/10.1145/331403.331405

[24] Silverstein, C., Marais, H., Henzinger, M., Moricz, M.: Analysis of a very large web search engine query log. SIGIR Forum **33**(1), 6–12 (1999). DOI 10.1145/331403.331405. URL `http://doi.acm.org/10.1145/331403.331405`

[25] Skobeltsyn, G., Junqueira, F., Plachouras, V., Baeza-Yates, R.A.: Resin: a combination of results caching and index pruning for high-performance web search engines. In: SIGIR, pp. 131–138 (2008)

[26] Soffer, A., Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y.S.: Static index pruning for information retrieval systems. In: SIGIR, pp. 43–50 (2001)

[27] Tzitzikas, Y., Spyratos, N., Constantopoulos, P.: Mediators over taxonomy-based information sources. The VLDB Journal **14**(1), 112–136 (2005)

[28] Vazirani, V.: Approximation algorithms. Springer (2001)

[29] Xie, Y., O'Hallaron, D.: Locality in search engine queries and its implications for caching. In: INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 3, pp. 1238–1247. IEEE (2002)

[30] Xie, Y., O'Hallaron, D.R.: Locality in search engine queries and its implications for caching. In: INFOCOM (2002)

[31] Yao, Y., Yao, B.: Covering based rough set approximations. Information Sciences **200**, 91–107 (2012)

[32] Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: WWW, pp. 387–396 (2008)

[33] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), 6 (2006). DOI http://doi.acm.org/10.1145/1132956.1132959

A Top-K SCRC: Proofs

Prop A.1 The computation of the values K_{ex} and K_{ro} is correct.

Proof:
(K_{ex})
Let $\langle e_1, e_2, e_3, e_4 \rangle$ be the ordering of document wrt their certain score. We start from the start and we proceed as long as $score_{cert}(e_i) \geq Missing^{up}$. Recall that $Missing^{up}$ is the maximum score that an unknown document, say e', can have. It follows that $score_{cert}(e_i) \geq score(e')$. Since $score(e_i) \geq score_{cert}(e_i)$, it follows that $score(e_i) \geq score(e')$. It is therefore obvious that $Set(top(K_{ex})(Ans_{cache}(q)))$ are certainly the K_{ex} most highly scored elements of $Ans(q)$.
(K_{ro})

We start from the start of the list and we proceed as long as $score_{cert}(e_i) \geq Found^{up}(B \setminus V)$ where V are the visited elements of the list so far. This means that we proceed as long the following inequalities hold:

$$\begin{array}{rcl} score_{cert}(e_1) & \geq & Found^{up}(\{e_2, e_3, e_4\}) \\ score_{cert}(e_2) & \geq & Found^{up}(\{e_3, e_4\}) \\ score_{cert}(e_3) & \geq & Found^{up}(\{e_4\}) \\ score_{cert}(e_4) & \geq & Found^{up}(\emptyset) \end{array}$$

Recall that $Found^{up}(X)$ is the maximum upper bound of the scores of the elements in X. So if the first inequality holds then this means that it is impossible that one of $\{e_2, e_3, e_4\}$ has a score that is greater than $score_{cert}(e_1)$. Let's assume that the first two (of the four) inequalities hold. They imply that $score(e_1) \geq score(e_2) \geq score(e_3)$. So the relative order of $\{e_1, e_2, e_3\}$ is correct, i.e. as in $Ans(q)_{|\{e_1,e_2,e_3\}}$.
◇

B Quality of the Approximation

Here we will discuss the quality of the approximation returned by the greedy algorithm. Let U be the query terms, and let $Rem(U)$ be the uncovered elements of U if we run an *exhaustive* exact set cover algorithm. If an exact set cover exists then obviously $|Rem(U)|$ = 0, and recall that this decision problem is NP-Complete. Let $Rem_G(U)$ be the uncovered elements of U if we run the *greedy* algorithm. Since we would like to cover all elements of U (for answering the incoming query only by cached queries), for evaluating the quality of the outcome of the greedy algorithm, we can use the following metric:

$$Z = \frac{|Rem_G(U)| - |Rem(U)|}{|U|}$$

Clearly, $Z = 0$ if either both algorithms return an exact cover, or none of them returns an exact cover but each of them leaves the same number of uncovered elements. If the greedy leaves more uncovered elements (than the exhaustive algorithm), then $Z > 0$. Obviously, 1 is the worst value for Z corresponding to the case where no elements of U are covered. However we will refine the upper bound later on.

Let $F = \{S_1, \ldots, S_k\}$ be a family of subsets of U (i.e. the lower queries in our problem). Now suppose that there is an exact cover (so the exhaustive algorithms returns YES and the particular exact cover). Below we will discuss the quality of approximation (using Z) of the greedy algorithm for various cases:

- **Singleton subqueries.** If for each $u \in U$, there is an S_i in F such that $S_i = \{u\}$ then $Z = 0$. The proof is trivial.
- **If** $|U| = 2$ **then** $Z = 0$**.** The proof is trivial (either F contains an identical query, or two singleton subqueries).
- **Case** $|U| = 3$**.** Here Z can be greater than zero. For example, consider $U = \{a, b, c\}$ and suppose $F = \{\{a, b\}, \{a\}, \{b, c\}\}$. An ESC exists, but the greedy could fail (if it selects $\{a, b\}$ at its first iteration). Here we have $Z = \frac{1-0}{3} = \frac{1}{3}$. In general we can say that an ESC of a set of terms $U = \{a, b, c\}$ can be one of the following:

  1. {{a,b,c}}
  2. {{a,b}, {c}}
  3. {{a}, {b,c}}
  4. {{a,c}, {b}}
  5. {{a}, {b}, {c}}

  In cases 1 and 5 the greedy algorithm will not fail. In cases 2, 3, 4 it can fail:
  case 2: if F contains the set {a,c},
  case 3: if F contains the set {a,b},
  case 4: if F contains the set {a,b}.
  It follows that in at most 3 out of the 5 cases, it can fail, and in each such case $|Rem_G(U)| - |Rem(U)|$ would be 1. It follows that the expected value of Z (if we assume that an ESC exists) is at least 2/5*0 + 3/5 * 1 = 3/5 = 0.6.
- **Case** $|U| \leq 3$**.** This is the most frequent case, as previous works on query length analysis have shown (Table 19 summarizes such results). It was also proved by our measurements; the analysis of the logs, that are presented in Section 7, shows that in 87% of the queries we have $|U| \leq 3$. Specifically, 27.7% are single word queries, 38% are two word queries, and 21% are three word queries. According to the aforementioned cases, and based on the

frequencies of the queries in our query logs, we get that the expected value of Z for queries with up to 3 words is: $E(Z) = P(|U| = 1)*0 + P(|U| = 2)*0 + P(|U| = 3)*0.6 = 0.277/0.87*0 + 0.38/0.87*0 + 0.21/0.87*0.6 = 0.144$.

- **The general case (including $|U| > 3$).** If an ESC exists then F contains at least one subset of U. In this case the greedy algorithm will select the biggest. Let $m = max|S_i|$ (note that $1 \leq m \leq |U|$). If the greedy fails to find any other set, then this means that $Z = (|U| - m - 0)/|U| = (|U| - m)/|U| = 1 - \frac{m}{|U|}$. It follows that this is an upper bound of Z for the general case.

Table 19: Query log analysis of previous works

Study	WSE	Date	$\|Q_e\|$	$P(\|U\| \leq 3)$
Jansen et al. [13], Table 6	Excite	1997	51,474	85%
Silverstein et al. [24], Table 2	Altavista	2 Aug - 13 Sep 98	993,208,159	87%
Jansen et al. [12], Table 7	AllTheWeb.com	2001	451,551	84.1%
Jansen et al. [12], Table 7	AllTheWeb.com	2002	957,303	84.9%
Xie and ÓHallaron [29], Figure1(a)	Vivisimo	14 Jan -17 Feb 2001	2,477,283	87%
Xie and ÓHallaron [29], Figure1(b)	Excite	20 Dec 1999	2,477,283	77%
Jansen and Pooch [11], Table 1	Fireball	1 July - 31 July 98	16,252,902	95.5%

C An Indicative Experimental Comparison with PLC

Here we report extra experimental results which demonstrate that SCRC is faster than the RC and the PLC. We used the same experimental setup as the one described in Section 7.6 and tested the performance of the SCRC using the same datasets over the Mitos WSE (Excite, Altavista). The same training sets were used for filling the PLC cache and the same test sets were used for submitting these queries to the Mitos WSE and evaluating its performance.

Each uncompressed posting list $I(t)$ of a term t in the posting lists cache (PLC) consists of pairs of the form $(d_i, tf_{d_i,t})$. We selected the terms to be cached, using the Q_{TFDF} scheme proposed in [5] which suggests caching the terms with the highest $\frac{pop(t)}{df(t)}$ ratio, where $pop(t)$ is the popularity of the term t in the evaluation queries. It has been shown that this scheme outperforms other schemes (i.e. caching the terms with the highest query-term frequencies [6], referred as Q_{TF} in [5]).

Figures 14 and 15 illustrate the average query response time of the Mitos WSE when employing the RC, the PLC and the SCRC over the *Altavista* set and the *Excite* set respectively. The first column depicts the average query evaluation time by using the answer of the index (no cache). The second, the third and the fourth column show the average response time of Mitos WSE when using the PLC, the RC and the SCRC respectively.

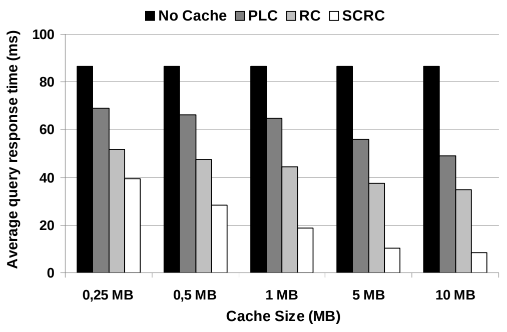


Figure 14: Average query response time over the *Altavista* query set (in ms)

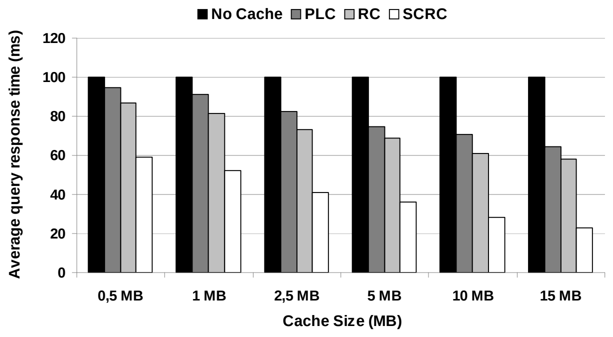


Figure 15: Average query response time over the *Excite* query set (in ms)

We observe that the RC is faster than the PLC and the SCRC outperforms the RC and the PLC in all cases. As the cache size increases, the speedup obtained by SCRC is significantly higher than the one obtained by the RC and the PLC.

In Figure 14, we observe that when the cache size is medium (i.e M = 1 MB), SCRC is at least 2 times faster than the RC and 3 times faster than the PLC. When the cache size is large (i.e *M = 10 MB*), SCRC is 4 times faster than the RC and 5 times faster than the PLC.

In Figure 15, we observe that when the cache size is medium (i.e *M = 5 MB*), SCRC is 2 times faster than the RC and the PLC. When the cache size is larger (i.e *M = 15 MB*), SCRC is 3 times faster than the RC and the PLC.

D Decomposability and Exact Set Cover

In the following example, we show why a plain set cover would lead to the wrong computation of the scores of the documents in the final answer $Ans(q)$ of a query q in a best-match retrieval model.

Example D.1 Consider a document collection that consists of only one document d, where d = "barack obama nobel prize", and that the WSE uses the varied *decomposable* scoring function of the Vector Space Model (the one described in Section 3 when ignoring W_q) for assigning scores to the matching documents. The score of a document d w.r.t the query q is given by:

$$Sim'_{cos}(d,q) = \frac{\sum_{t \in t(d) \cap t(q)} w_{d,t} \cdot w_{q,t}}{W_d}$$

Assume queries q_{c_1}="barack obama" and q_{c_2}="obama nobel prize".

Let us compute the score of the document d for each of these queries. The vector of the document d is $\overrightarrow{d} = \{1,1,1,1\}$ and the vectors of the queries q_{c_1} and q_{c_2} are $\overrightarrow{q_{c_1}} = \{1,0,1,0\}$ and $\overrightarrow{q_{c_2}} = \{0,1,1,1\}$ respectively. The score of the document d w.r.t to queries q_{c_1} and q_{c_2} is:

$$Sim'_{cos}(d,q_{c_1}) = \frac{1*1+0*0+1*1+0*0}{\sqrt{2}} = \frac{2}{\sqrt{2}} = \sqrt{2} = 1.41$$

$$Sim'_{cos}(d,q_{c_2}) = \frac{1*0+1*1+1*1+1*1}{\sqrt{2}} = \frac{3}{\sqrt{2}} = 3*\frac{\sqrt{2}}{2} = 2.12$$

Hence, $Ans(q_{c_1}) = \{(d, 1.41)\}$ and $Ans(q_{c_2}) = \{(d, 2.12)\}$.

Assume that the query q= "barack obama nobel prize" is submitted to the WSE. Then, $C = \{q_{c_1}, q_{c_2}\}$ is a set cover of $t(q)$, since $t(q_{c_1}) \bigcup t(q_{c_2}) = t(q)$. However, C is not an exact set cover, since $t(q_{c_1}) \cap t(q_{c_2}) \neq \emptyset$. We now show why the score of the document d derived through the set cover C is not equal to $Sim'_{cos}(d,q)$. The vector of the query q is $\overrightarrow{q} = \{1,1,1,1\}$ and the score of the document d w.r.t to q is computed as:

$$Sim'_{cos}(d,q) = \frac{1*1+1*1+1*1+1*1}{\sqrt{2}} = \frac{4}{\sqrt{2}} = 4\frac{\sqrt{2}}{2} = 2\sqrt{2} = 2.852$$

Now we compute the score of the documents in $Ans(q)$, which is the union of the documents in the answers of the queries in C. Hence, $Ans(q) = d$.

The score $Score(d, q)$ of a document $d \in Ans_C(q)$ is the sum of the scores that d received in the answers of C. Hence, we have that:

$$Score(d, q) \quad = \quad \sum_{q_c \in C} Score(d, q_c) = 1.41 + 2.12 = 3.53 \neq Sim'_{cos}(d, q) \diamond$$

$\diamond$