University of Crete

Computer Science Department

Set Cover-based Results Caching for Best Match Retrieval Models

Papadakis Myron

Master's Thesis

Heraklion, February 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Προσωρινή Αποθήκευση (Caching) αποτελεσμάτων για Μοντέλα Ανάκτησης Βέλτιστου Ταιριάσματος βασισμένη στην Κάλυψη Συνόλων

Εργασία που υποβλήθηκε από τον
Μύρωνα Εμμ. Παπαδάκη
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Μύρων Παπαδάκης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Ιωάννης Τζίτζικας, Επίκουρος καθηγητής, Επόπτης

Βασίλης Χριστοφίδης, Καθηγητής, Μέλος

Ευάγγελος Μαρκάτος, Καθηγητής, Μέλος

Δεκτή:

Πάνος Τραχανιάς, Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Φεβρουάριος 2010

Set Cover-based Results Caching for Best Match Retrieval Models

Papadakis Myron

Master's Thesis

Computer Science Department, University of Crete

Abstract

Modern Web Search Engines (WSEs) employ various techniques in order to reduce response times, processing load and hardware requirements. Query evaluation in WSEs requires processing (retrieving and ranking) a large amount of data of its underlying index, especially when query terms have long posting lists. One of the most effective techniques for enhancing the performance of a WSE is the use of caching, which can be divided into two main categories: Results Caching (for short RC) and Posting List Caching (for short, PLC). Results caching allows answering queries that have been previously submitted at almost no cost, without accessing the main index of the WSE, since future requests for the same queries are served immediately by the cache. On the other hand, posting lists caching promises higher cache hit ratios but it does not avoid the query evaluation costs.

In this thesis, we propose and experimentally evaluate a novel results caching technique, called SCRC (Set Cover Results Cache), which bridges the gap between results caching and posting lists caching: identical queries are captured as in plain results caching while combinations of cached sub-queries are exploited as in posting lists caching and therefore SCRC can be considered as a generalization of posting lists caching. We reduce the problem of finding the appropriate cached sub-queries, to the exact set cover problem, and we adopt a greedy algorithm for solving it approximately. The correctness of this query evaluation approach is guaranteed if the scoring function of the retrieval model is additively decomposable and we prove that several best-match retrieval models (e.g VSM, Okapi BM25 as well as hybrid retrieval models) which are traditionally used in Information Retrieval and WSEs rely on such scoring functions.

To estimate the practical impact of our approach, we analyzed streams of queries submitted to real world WSEs (Excite, AllTheWeb, Altavista) with metrics that we defined and our findings indicate that approximately 35% of these queries can be formulated as a disjoint union of the rest submitted queries. Subsequently, we proposed and evaluated several cache

filling strategies for maximizing the speed up obtained by the SCRC. Furthermore and since most users examine only the first pages of results, we proposed a variation of SCRC called Top-K SCRC, which stores only the top-K results of each cached query, and we defined metrics for characterizing the quality of the composed top-K answer.

The above techniques were comparatively evaluated over the Mitos WSE and over two versions of its index: one based on an index represented in an Object-Relational DBMS and another based on an Inverted File. The results over this indices showed that in best-match retrieval models the SCRC achieves a speedup which is two times higher than the one obtained by the RC and three times higher than the one obtained by the PLC.

Supervisor: Yannis Tzitzikas

Assistant Professor

Προσωρινή Αποθήκευση (Caching) αποτελεσμάτων για Μοντέλα Ανάκτησης Βέλτιστου Ταιριάσματος βασισμένη στην Κάλυψη Συνόλων

Παπαδάκης Μύρων
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Οι μηχανές αναζήτησης του παγκόσμιου στού χρησιμοποιούν διάφορες τεχνικές για να μειώσουν τους χρόνους απόκρισης, το κόστος της επεξεργασίας επερωτήσεων και τις απαιτήσεις σε υλικό. Γενικά, η αποτίμηση μιας επερώτησης απαιτεί την επεξεργασία (ανάκτηση και διαβάθμιση) μεγάλου όγκου δεδομένων του ευρετηρίου της μηχανής, ιδιαίτερα όταν οι όροι της επερώτησης έχουν μεγάλες λίστες εμφάνισης. Ένας από τους πιο αποτελεσματικούς τρόπους βελτίωσης της απόδοσης μιας μηχανής είναι η χρήση τεχνικών caching οι οποίες διακρίνονται σε δύο μεγάλες κατηγορίες: caches αποτελεσμάτων (Result Caches, για συντομία RC), και caches όρων ευρετηρίου (Posting Lists Caches, για συντομία PLC). Μια RC μας επιτρέπει να απαντάμε πολύ γρήγορα - και χωρίς να χρειάζεται να ανατρέξουμε στο ευρετήριο - επερωτήσεις που έχουν υποβληθεί αυτούσιες στο παρελθόν. Από την άλλη, μία PLC επιτυγχάνει υψηλότερο ποσοστό επιτυχών επισκέψεων (hits), ωστόσο δεν αποφεύγει το κόστος της αποτίμησης των επερωτήσεων.

Η παρούσα μεταπτυχιακή εργασία εισάγει μια νέα τεχνική caching που ονομάζεται SCRC (Set Cover Results Cache) η οποία συνδυάζει τα ατού της RC και της PLC. Συγκεκριμένα, για κάθε επερώτηση που υποβάλλεται εάν η SCRC περιέχει μια ισοδύναμη επερώτηση τότε η απάντηση της επιστρέφεται άμεσα (όπως σε μια ResultCache), διαφορετικά επιχειρείται αποτίμηση συνδυάζοντας τις απαντήσεις άλλων cached (υπο-)επερωτήσεων, εκ τούτου αποτελεί γενίκευση της PLC. Ανάγουμε τη διαδικασία εύρεσης των πιο κατάλληλων cached υποερωτήσεων στην επίλυση του προβλήματος 'Ακριβούς Κάλυψης Συνόλων' (Exact Set Cover Problem) και χρησιμοποιούμε έναν άπληστο (greedy) αλγόριθμο για την προσεγγιστική επίλυση του. Η ορθότητα των αποτελεσμάτων που προκύπτουν από αυτή την διαδικασία αποτίμησης επερωτήσεων, εξαρτάται από το εάν η συνάρτηση βαθμολόγησης είναι αθροιστικά αποσυνθέσιμη (additively decomposable) , και δείχνουμε ότι αρκετά μοντέλα ανάκτησης που παραδοσιακά

έχουν χρησιμοποιηθεί στην Ανάκτηση Πληροφορίας και στις μηχανές αναζήτησης (π.χ. διανυσματικό μοντέλο, Okapi BM25 καθώς και υβριδικά μοντέλα ανάκτησης) βασίζονται σε τέτοιες συναρτήσεις διαβάθμισης. Για να εκτιμήσουμε την αποδοτικότητα της προτεινόμενης cache σε πραγματικές συνθήκες, αναλύσαμε ροές επερωτήσεων (query logs) πραγματικών μηχανών αναζήτησης (Excite, Altavista, AllTheWeb) με μέτρα που ορίσαμε, και τα αποτελέσματα έδειξαν ότι περί το 35% των επερωτήσεων μπορούν να εκφραστούν ως ένωση όρων άλλων υποβεβλημένων επερωτήσεων. Βάσει αυτών των στοιχείων προτείναμε και αξιολογήσαμε τεχνικές γεμίσματος της cache για αύξηση της επιτάχυνσης που επιτυγχάνει η SCRC. Επιπροσθέτως, και για να αξιοποιήσουμε το γεγονός ότι η πλειονότητα των χρηστών εξετάζει μόνο τις πρώτες σελίδες αποτελεσμάτων, ορίσαμε και μια παραλλαγή της SCRC, την Top-K SCRC, η οποία αποθηκεύει μόνο τα K κορυφαία αποτελέσματα των απαντήσεων και ορίσαμε μέτρα για τον χαρακτηρισμό της ποιότητας των Top-K απαντήσεων που προκύπτουν από τον συνδυασμό υποερωτημάτων.

Οι παραπάνω τεχνικές αξιολογήθηκαν συγκριτικά στη μηχανή αναζήτησης Μίτος και σε δύο εκδόσεις του ευρετηρίου της: μία που βασίζεται σε ένα Αντικειμενο-Σχεσιακό Σύστημα Διαχείρισης Βάσεων Δεδομένων (Object-Relational DBMS), και μία που βασίζεται σε ανεστραμμένα αρχείο (inverted file). Τα αποτελέσματα έδειξαν ότι σε μοντέλα βέλτιστου ταιριάσματος η SCRC επιτυγχάνει τη διπλάσια μέση επιτάχυνση σε σχέση με τη RC, και την τριπλάσια μέση επιτάχυνση σε σχέση με την PLC.

Επόπτης Καθηγητής: Γιάννης Τζίτζικας
Επίκουρος Καθηγητής

Στην οικογένεια μου

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επόπτη καθηγητή μου κ. Γιάννη Τζίτζικα που μου έδωσε την ευκαιρία να ασχοληθώ με ένα επίκαιρο και πολύ ενδιαφέρον θέμα καθώς και για την ουσιαστική συμβολή του στην ολοκλήρωση της παρούσας εργασίας. Θέλω επίσης να ευχαριστήσω τους καθηγητές κ. Βασίλη Χριστοφίδη και κ. Ευάγγελο Μαρκάτο για την προθυμία τους να συμμετάσχουν στην εξεταστική επιτροπή καθώς και για τις εύστοχες παρατηρήσεις τους που συντέλεσαν στην βελτίωση της εργασίας αυτής.

Τα query logs διαφόρων μηχανών αναζήτησης τα οποία χρησιμοποίησα στην εργασία αυτή, μου τα παρείχε ο Jim Jansen τον οποίο και ευχαριστώ θερμά. Ευχαριστώ τον Παναγιώτη Παπαδάκο για την τεχνική υποστήριξη και την βοήθεια του στην μηχανή Μίτος. Ένα μεγάλο ευχαριστώ χρωστάω σε όλους τους φίλους και συναδέλφους κατά την διάρκεια των σπουδών μου. Ευχαριστώ ιδιαίτερα τον Γιώργο, τον Παντελή, τον Θάνο και την Εύη με τους οποίους μας δένει μία μακρόχρονη σχέση φιλίας, για την υποστήριξη τους και τις ευχάριστες στιγμές που περάσαμε μαζί. Κυρίως όμως θέλω να ευχαριστήσω την Κατερίνα για την κατανόηση της, την υπομονή της και την συνεχή συμπαράσταση της στον δύσκολο αυτό αγώνα.

Κλείνοντας, το μεγαλύτερο ευχαριστώ το χρωστάω στους γονείς μου Μανώλη και Γεωργία και στα αδέρφια μου Νίκο, Ελένη και Μαρία για την αγάπη τους και την υποστήριξη τους όλα αυτά τα χρόνια.

Η παρούσα μεταπτυχιακή εργασία είναι αφιερωμένη στην οικογένεια μου.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern Web Search Engines (WSEs) receive millions of queries in a daily basis from different users therefore they are usually implemented on thousands of clusters and employ various techniques in order to reduce response times, processing load and hardware requirements.

In order to make clear the major query processing costs associated with the evaluation of a query, in this chapter we describe in brief the structure of the typical inverted index and the typical search process in a WSE.

1.1 Inverted File

WSEs and information retrieval systems in general, 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 and consists of the *vocabulary* T and the *occurrences* (*posting file*). The vocabulary contains all the distinct terms which are extracted from the crawled documents. For each term $t \in T$, the vocabulary also stores the number of documents that contain t ($df(t)$) and a pointer to the start of the corresponding posting list. Each term $t_i \in T$, $i = \{0, 1, \ldots, m\}$ is associated with a *posting list* $I(t_i)$. The document frequency $df(t)$ of a term t corresponds to the length of its inverted list. A posting list contains a set of document entries $\{\langle d_1, tf_{d_1,t}\rangle, \langle d_2, tf_{d_2,t}\rangle, \ldots, \langle d_n, tf_{d_n,t}\rangle\}$, where d_n is the largest document identifier in the set of postings, since entries are usually sorted by increasing document order. Each entry $\langle d, tf_{d_i,t}\rangle$ in $I(t)$ contains the identifier of each

document d_i that contains t (a unique serial number, known as the document identifier, *docID*) and optionally additional information, such as the frequency of the term in the document ($tf_{d_i,t}$) and the positions of the occurrences. The occurrences stored in the posting file in turn refer to entries in the document file, which is also kept in secondary storage. The set of all these lists $I = \{I(t_0), I(t_1), \ldots, I(t_m)\}$, $m = |T|$ forms the *inverted index*.

Figure 1.1 shows the structure of a typical inverted index. The terms in the vocabulary are sorted alphabetically and each posting list is sorted by the document identifier.

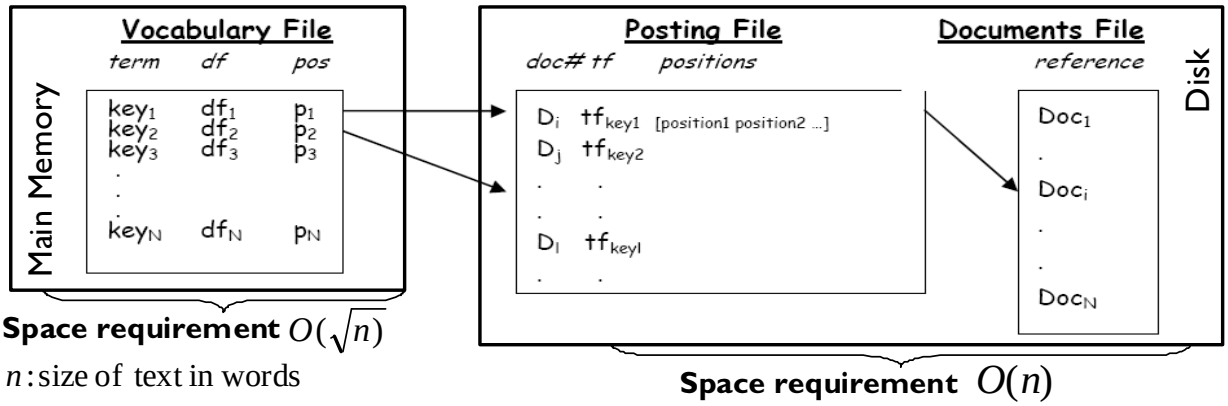


Figure 1.1: Inverted Index

The space required for the vocabulary is small and it can be retained in the main **memory**. According to *Heap's law* [23, 8] which estimates the vocabulary size as a function of collection size, the vocabulary grows as $\mathcal{O}(n^{\beta})$, where β is constant between 0.4 and 0.6 and n the size of the text in words. Hence, the space required for the vocabulary is $\mathcal{O}(\sqrt{n})$.

However, the occurrences are too large to fit the main memory and they are normally stored on the **disk**. Since each word appearing in the text is referenced once in that structure (if positions are kept), the extra space is $\mathcal{O}(n)$.

1.2 The problem

Assume that a user submits a query $q = \{t_0, t_1, \ldots, t_n\}$ to the WSE.

- **Query Pre-processing**: The query is decomposed into its distinct terms. Frequent words which are not useful for retrieval (i.e articles), namely *stopwords* are usually

discarded, while all letters are reduced to lower case. Moreover, *stemming* of the query terms eliminates grammatical variations of the same word by reducing it to a word root (stem). For example, a stemming algorithm reduces the words "fishing", "fished", "fish", and "fisher" to the root word "fish".

- **Vocabulary Search**: Each term $t_i \in q$ is fetched from the vocabulary.
- **Retrieval of occurrences**: The posting lists $I(t_0),\ldots I(t_n)$ of the query terms are retrieved from the posting file.
- **Manipulation of occurrences**: The posting lists of the terms are then processed i.e to extract the matching documents and merge the posting lists, to measure the relative importance of terms in both the indexed documents as in the queries.
- **Document Scoring**: The scoring function of the retrieval model (i.e Okapi BM25, VSM) that the search engine uses, computes and assigns scores to the matching documents in order to indicate their relevance (closeness) to the given query and rank the results. The higher the similarity score assigned to a document, the greater the estimated likelihood that a human would judge it to be relevant.
- **Sorting and presenting results to the user**: Matching documents are sorted in decreasing order with respect to their assigned score. The document identifiers are then used to retrieve the corresponding titles and URLs of these documents from the document index. For each document a small summary of the document is generated (called *snippet*) in order to give a succinct explanation to the user of why the document matches the query.

Ranking is based on scoring functions and in most best match retrieval models (i.e Okapi BM25, VSM) every document containing at least one of the query terms is considered a candidate and is assigned a score indicating its similarity with the incoming query, hence a query may match a huge number of documents. Moreover, ranked queries are usually expressed in natural language and can therefore contain a large number of terms. Adding more terms results in more disk accesses. Hence, even for a single query the WSE has to access the index, traverse a large number of inverted lists and process a large amount of

various data, especially for terms with long posting lists, which increases the response time of the WSE.

Caching is a well known technique that can be used to accelerate the access to frequently used data. Accessing data in memory is much faster than accessing data on disk. In many contexts, such as Web Search Engines (WSEs) a significant fraction of user queries and query terms are very popular and are re-submitted many times by different users. A search engine can take advantage of this behavior by caching these frequently accessed data. Results caching (caching queries and their answers) and posting lists caching (caching terms and their posting lists) are the state of-the-art caching techniques in WSEs. In this thesis, we introduce a new caching scheme, namely *SCRC* which leverages the advantages of the former approaches. The core idea of the SCRC, is to exploit cached sub-queries for answering the incoming queries, if there is not an identical query in the cache. For example, consider that we have cached the answers of the queries q_1="barack obama", q_2="facebook", q_3="popularity" in a results cache (RC or SCRC) or the terms of these queries and their corresponding posting lists in a posting lists cache (PLC). Then, in a SCRC we can answer from the cache queries of the form: "barack obama popularity", "barack obama facebook", and "barack obama facebook popularity", and we can also speedup the evaluation of queries like "barack obama inauguration". A results cache cannot answer any of the former queries, since there are not any identical cached queries. A posting lists cache can also answer the former queries but at the expense of a higher cost, since their query evaluation cannot be avoided.

1.3 Contribution of this thesis

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

- a survey of the related work of caching in web search engines
- a novel results caching scheme that leverages the advantages of results caching and posting lists caching and is based on cached subqueries
- the notion of *decomposable scoring functions* which determines the applicability of the proposed caching scheme and an analysis of several best-match retrieval models (i.e

VSM, Okapi BM25 and hybrid retrieval models) that rely on such scoring functions

- a thorough analysis of query logs of real WSEs which shows that the proposed approach achieves up to 30% higher hit rates than a plain results cache
- several static cache filling policies, which aim at maximizing the hit rate obtained by a SCRC that caches the complete answers of the queries
- a variation of the SCRC (Top-K SCRC) where only the top-K answers are stored in the cache and metrics for measuring the accuracy of answers derived through set cover hits
- a thorough comparative experimental results over real query sets which shows that a Top-K SCRC is 2 times faster than a plain Top-K RC

1.4 Organization of this thesis

This thesis is organized as follows.

Chapter 2 discusses related work and provides an overview of the current state-of-the-art approaches for caching in Web Search Engines.

In Chapter 3, we introduce a novel caching scheme for results caching, namely SCRC. We explain how the set cover problem relates to query evaluation in WSEs and we describe the complete evaluation process in the presence of a SCRC.

In Chapter 4, we provide a thorough analysis of query logs of real WSEs (*Excite*, *AllTheWeb*, *AltaVista*) and show the benefits of set cover-based results caching in WSEs. We evaluate the proposed approach over the Mitos search engine and present our results.

Chapter 5 concludes this thesis and identifies issues that are worth further research.

Chapter 2

Related Work

2.1 Caching in Web Search Engines

Caching of results of search engines as an optimization technique was firstly noted by Brin and Page [14] in their description of the prototype *Google* search engine. There are several kinds of information that can be cached by a Web search Engine. Results caching [34, 47, 31, 21, 7, 44, 22, 42, 32, 6] and posting lists caching [48, 42, 32, 7, 6] are the state of the art caching techniques in current Web Search Engines. Independently of what the cached elements are (query results or posting lists), caching policies can be either *static* or *dynamic*.

In this section we provide an overview of the related work. We have to note that there is a large body of work that has been devoted to query optimization techniques for improving the performance of WSEs, such as *early termination techniques* [39, 3, 2] and *index pruning* [45, 12, 13, 20, 35]. However, these works do not consider caching and they are not considered implicitly relevant to the scope of this thesis. Instead, they can be considered as complementary to caching in search engines.

2.1.1 Results Caching

Caching user queries and their answers, namely *results caching (RC)*, is one of the most popular and effective techniques for improving the efficiency of large scale WSEs. Motivation for caching query results arises from the significant locality in user queries [34, 47]. About

30% to 40% of queries are repeated queries that have been re-submitted in the past and are shared by many different users [47]. Moreover, it has been shown that the query repetition frequency follows a Zipf distribution [47]. Query results caching can be implemented at various points of the network, such as the client side, on a proxy or on the server side. However, due to the high level sharing of popular queries among different users in web search engines most studies so far for caching results in WSEs have focused on the design and implementation of server-side caches [34, 29, 21].

A results cache stores usually the top-K results (or even all the results) of previously submitted queries. Consider that even a single term query may involve millions of relevant documents in its posting list. Caching queries and their pre-computed results avoids totally query evaluation. A lot of queries can be filtered out by a results cache, thus avoiding the expensive I/O operations of query evaluation. There are several formats that a cached answer may have. A simple format of an answer comprises the title, the URL and a snippet for each matching document (*html cache*). Alternatively, a results cache can store only the document identifiers of the matching documents for each query (*docId cache*) in order to allow the accommodation of greater number of cache entries. Compression techniques can be used to further reduce the size of the cache. However, the latter format implies that additional processing has to be payed for presenting the results in a consistent way to the user. The major weakness of plain results caching is that it speeds up the query evaluation of a query q if and only if it contains an *identical* query $q' \equiv q$ and the results pertaining to it.

Static Caching: *Static results caching* exploits the locality of reference in a query stream and it is based on historical information. A static results cache is usually initialized by the most frequent queries, using the query log of the search engine. The cache content remains unchanged until the next update of the cache. However, it should be periodically updated in order to retain high hit rates. The rationale behind static caching is the presence of high locality of queries submitted by users to a search engine. Moreover, static caching is robust over time, because the distribution of frequent queries changes very slowly [6]. Despite the fact that query frequency is the major feature for filling a static results cache, several other features have been proposed [36, 7, 22].

Dynamic results caching: In a *dynamic results cache*, the cache content changes dynamically with respect to the query traffic. Several queries retain popular just for a short time interval. Consider for example that a very significant event occurs, such as a big earthquake or a terroristic attack (e.g 11 September). Queries submitted acquiring information about this event will be very frequent, since users will be interested in finding out more information, but these queries will probably be very popular just for a short period. Static caching cannot address sudden changes in user queries. Dynamic caching addresses this problem, by replacing cache entries according to the sequence of requests (queries). Central to the success of any dynamic cache is its eviction policy; the cache must decide which cache entry it must evict upon a cache miss. A very simple but very effective strategy in dynamic caching which has been traditionally used in WSEs is to evict the least recently used (LRU) item from the cache.

To leverage the advantages of static and dynamic results caching, a results cache can be split into a static and a dynamic segment [21, 7].

Previous Work on Caching in WSEs: In one of the earliest works regarding caching in WSEs, Markatos [34] introduces caching query results as a technique to reduce the query response time of a search engine. Using a log of almost one million queries submitted to the *Excite* WSE, he studies query log distributions and shows the existence of high temporal locality in queries. He proposes that either static or dynamic caching at the search engine level can be beneficial for WSEs. He compares the hit ratio obtained by applying a static policy which selects the most frequent queries of a stream and four dynamic replacement policies - LRU (Least Recently Used) and three variations (FBR, LRU/2, SLRU). He demonstrated that warm, large caches of search results can attain hit ratios of close to 30%. Moreover, he showed that static query result caching is a good choice for small cache sizes, but dynamic caching is better for large cache sizes.

Luo et al. [33] also employ caching of result pages, and combine the answers of previously submitted queries in order to form the answers of larger queries. However, their approach is not realistic in modern WSEs, since they assume a boolean retrieval model. Current WSEs use more sophisticated scoring functions for assigning similarity scores to matching documents and each query term is assigned a non-binary weight, which in practice has a

great impact on the ordering of the final results.

Fagni et al. [21] employ an hybrid results caching scheme, namely the *Static Dynamic Cache (SDC)*. The available cache space is split into two segments: a static segment and a dynamic segment. The static segment is a readonly static results cache which keeps the most frequent queries extracted from a past query log. The dynamic segment initially contains queries that could not fit the static part of the cache and is used for capturing changes in the query traffic. The rationale behind the SCD policy is addressing both frequent and recent queries. Regarding the dynamic segment, the authors experimented with several cache replacement policies (LRU, SRLU, FBR, 2-queue, PDC) and showed that there is no need for adopting a complex eviction policy, since the most popular queries are already captured by the static cache and therefore the LRU policy is the best choice for the dynamic set, due to its simplicity. Authors also adopted an adaptive prefetching strategy along with the SDC cache, and showed that devoting a large fraction of entries to the static segment of the cache along with prefetching obtains the best hit rate.

R. Yates et al. [7] propose a general cache management policy for caching query results, which is fully dynamic in contrast to the SDC cache [21]. They suggest the use of admission policies in order to detect infrequent queries that will not probably be submitted in the future and prevent them from preserving any cache space. Query characteristics are used in order to estimate the benefit of a query. This policy uses two features based on the query usage, the length of the query and the number of non-alphabetical characters in the query, and one feature which is based on the frequency of the queries. The intuition is that long queries or queries with many non-alphanumerical characters are not likely to be popular. The usage of the past frequency of the queries gave higher hit rates opposed to using stateless features.

2.1.1.1 Caching and Prefetching of query results

In the context of enhancing the performance of a Web search engine, several works have studied in addition to results caching, prefetching of the query results in anticipation of user requests [30, 29, 21]. Hence, in addition to storing results that are requested by users in the cache, search engines may also prefetch additional pages of results (one page of results usually contains 10 results) in order to prepare the cache to answer possible requests for

following pages. Studies on user web search behavior report that at least 58% of the users view the top-10 results, and that no more than 12% of the users view through more than 3 result pages [26, 43]. Moreover, at least 15% of the users ask for the second page of results within a short time after submitting the query. Hence, when a user submits a query the second and even the third page of results can be also cached, apart from the first one in order to further increase the cache hit rate.

Lempel and Moran [29], introduce an improved caching policy for caching search engine results, namely PDC (Probabilistic Driven Caching) which estimates a probability distribution of all the possible queries that can be submitted to a WSE. They also introduce prefetching for pages of query results that are likely to be requested shortly (i.e., when a user requests a second or third page of results). The authors studied query log features and the length of the search sessions over a log of seven million queries of the AltaVista WSE, in order to assess the behavior of user during a search session. They showed that the frequencies of queries in their log followed a power law and that prefetching of search engine results can increase hit ratios by 50% for large caches and can double the hit ratios of small caches.

2.1.1.2 Cost-aware results caching policies

Most of the previous works that we reported, have focused on maximizing the cache hit ratio and the cost of each cached query is considered the same. Alternate approaches have been proposed that consider the costs of evaluating each query and aim at maximizing the benefits from caching, rather than maximizing the cache hit ratio [17, 22, 1]. In this context, storing a query in the results cache considers the cost of computing its answer and the success of the caching policy is measured through the achieved cost savings rather than the cache hit rate.

Gan and Suel [22] study weighted results caching techniques, where the cost of evaluating a query is estimated by the sum of the posting lists of its query terms, implying that the dominant cost of its evaluation is the fetching of its posting lists from the index. They propose eviction policies, which address both the frequency and the cost of each query. Following the work of Fagni et al. [21], their experimental results show that a combination of a static and a dynamic cache is the most efficient cache strategy. The best results were

obtained when devoting 80% of the total cache space in a static set in which the score for each query is calculated as the product between the query frequency and an estimation of the cost of solving the query, and 20% of the space to the dynamic set. The dynamic part is implemented using the Landlord policy which assigns in each cached query a deadline which is proportional to the cost of evaluating it. In the context of feature-based caching, they extend the work of B.Yates et al. [7] and propose a set of new features (e.g. the query length, the average number of clicks of each query) apart from the query frequency that are likely to be considered useful in results caching.

Altingovde et al. [1] propose a cost-aware caching strategy for the static caching of the query results in WSEs. The cost of each query is determined by its execution time, which includes decompressing the postings of the query terms, computing the query and document similarities and determining the top-K document identifiers in the final answer set. The profit of each query is determined by the product of its frequency and its cost. The experimental evaluation over a real log showed a reduction in the total query processing times up to 3%.

2.1.2 Posting Lists Caching

An alternative approach is to cache individual query terms and their corresponding posting lists, namely *postings lists caching (PLC)*. Posting lists caching promises higher hit rates than results caching, since terms repeat more significantly than queries [6]. On the other hand, posting lists need more space, especially in huge document collections.

Typically, Web search engines dedicate an amount of their available memory to a cache of posting lists, in order to retain terms that are frequency queried and reduce disk accesses for fetching posting lists from the index during query evaluation. Posting lists caching (or index caching) was firstly noted by Jónsson et al. [27] and by Brown et al. [16] but with a different setup from that of current search engines.

Posting lists caching implies storing in main memory parts of the inverted index that are frequently accessed and it is more advantageous in terms of cache utilization, since the cached query terms can be combined for answering the incoming query. In practice, the terms and their posting lists form a sort of a small in memory inverted index. Moreover, these data can be kept compressed in order to reduce memory space and allow a larger

fraction of terms and their corresponding posting lists to fit in the cache, resulting in higher hit rates [48, 5]. Zhang et al. [48] combine various index compression techniques and list caching and perform an evaluation of several inverted list compression algorithms. They show that the benefits of this combination depends actually on the machine parameters such as disk transfer rate, main memory, and CPU speed. B. Yates et al. [5] studied the benefits of keeping compressed the postings in a posting list cache. Their results in a real system showed that compression is always beneficial, resulting in lower response times of the WSE.

Since some terms are more beneficial than the others and due to memory limitations a posting lists cache cannot retain the posting lists of all the terms (assuming that the size of the main index is larger than the available main memory), several algorithms have been proposed for selecting the most "profitable" terms and their full posting lists. As in result caches, caching policies for posting lists can be either *static* or *dynamic*. Since posting lists have varying length, caching them dynamically is not very efficient, due to the complexity in terms of efficiency and space [6]. Efficient static caching policies have been studied by B.Yates et al. in [6], [9]. In [9], authors consider as the most profitable terms, the terms with the highest frequency as extracted from the WSE log (Q_{TF} algorithm).

In [6], authors propose a static caching algorithm of terms, called Q_{TFDF} which favors terms with high frequency and short posting lists. The profit of a term t is proportional to its occurrences in the query stream Q_e and inversely proportional to the space occupied by its posting lists. Moreover, this algorithm outperforms previous static caching policies (i.e Q_{TF}) and it is more effective than dynamic caching with, for example, LRU, LFU or the dynamic version of the Q_{TFDF}. They showed that the drawback of the Q_{TF} algorithm is that most of the frequent terms tend also to have long posting lists, and as a result, few posting lists fit in cache. Authors also experimented with hybrid policies for combining the advantages of static caching and dynamic caching of terms. Their results by using hybrid policies, indicate that when the cache size is small their is a slight improvement in the cache hit rate. However, as the cache size increases the performance of the Q_{TFDF} algorithm levels the performance of the hybrid policy.

2.1.3 Multi-level Caching

To leverage the advantages of results caching and posting lists caching, *hybrid caches* have been proposed [42, 32, 9, 6].

A *two-level caching* system is proposed by Saraiva et al. [42] for caching query results and inverted lists. The first level of the cache, caches frequent queries and their corresponding query results, using the LRU eviction policy, while the second level caches the posting lists of the query terms in order to avoid disk latency. Both cache levels are dynamic. Overall, their caching strategy resulted in a threshold increase in the throughput of the system by a factor of 3, while preserving the response time per query.

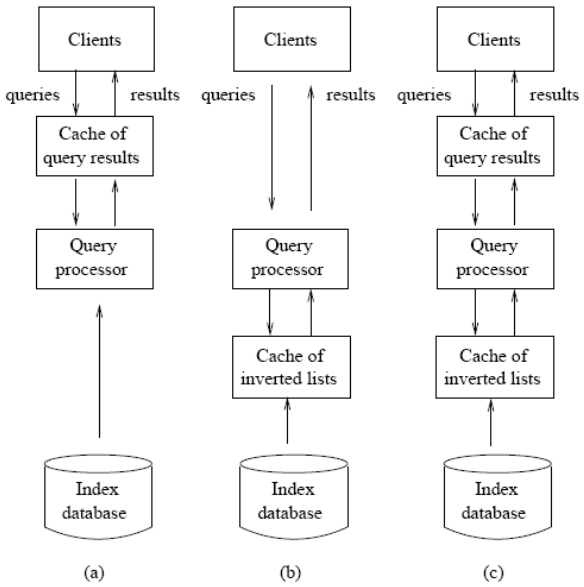


Figure 2.1: Search engine cache implementations (a) query results, (b) inverted lists and (c) two-level (From Saraiva et al. [42])

B.Yates and Jean [9] propose a similar organization which suggests a results cache holding the precomputed answers of the most popular queries and a static posting lists cache retaining the posting lists of the most popular query terms. Their results showed 7% reduction in the query evaluation time.

Long and Suel [32] extend the work of Saraiva et al. [42] and propose a three level caching architecture to improve the query throughput of a search engine. The first level of the cache stores previously computed query results, the second level of the cache stores

inverted lists of popular terms, while the third level contains frequently occurring pairs of terms and stores the intersection of their corresponding inverted lists on the disk which are not captured by result and list caches in 2-level architectures. In their results they report a 75% reduction in disk blocks read at query time, with a cache equal to 2.5% of index size. Figure 2.2 shows the architecture of the three level cache.

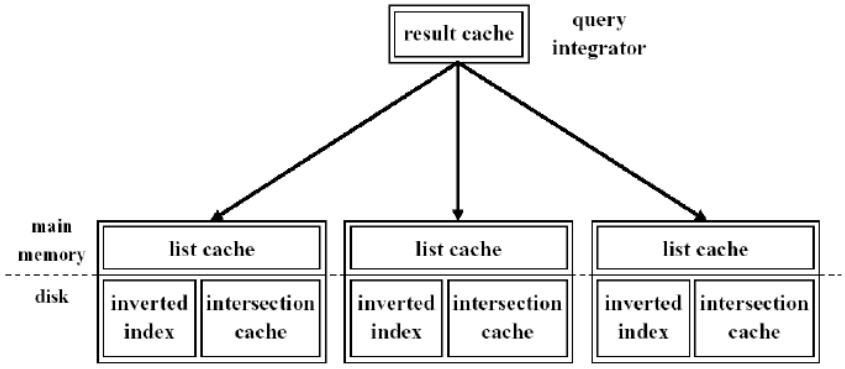


Figure 2.2: Three-level caching architecture with result caching at the query integrator, list caching in the main memory of each node, and intersection caching on disk. (From Long and Suel [32])

The impact of different caching policies in WSE is explored in [6]. Authors use a Yahoo! query log to compare the effectiveness of results caching versus posting lists caching. They show that posting lists caching offers higher hit ratios than results caching, and that static caching of terms can be more effective than dynamic caching. They conclude that in order to achieve optimal performance a cache should be divided into two parts, a results cache and a posting lists cache, and in this context they try to find the optimal division of the cache.

Skobeltsyn et al. [44] propose an architecture that combines results caching with index pruning to reduce the query processing load of a WSE. Figure 2.3 illustrates this architecture. Each query is firstly filtered out by the results cache. Upon a cache miss, the pruned index is asked to answer the query. In case the pruned index cannot answer it, the main index is in charge of query processing. Authors investigated both *term* and *document pruning* approaches. The most interesting result from this study is how optimization techniques that are employed by search engines impact each other. They show how results caching impacts the query load that needs to be served from the main index, and therefore changes

the effectiveness of index pruning, which attempts to serve some queries from a pruned index.

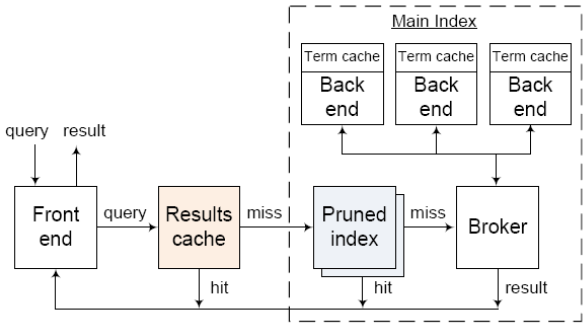


Figure 2.3: Query processing scheme with the ResIn architecture. (From Skobeltsyn et al [44])

2.1.4 Synopsis and Discussion

Summarizing, caching query results achieves lower hit ratios than posting lists caching, since it is limited on answering only identical queries but it can be more efficient in terms of performance, especially when query processing costs are high. Moreover, static caching is robust over time, because the distribution of frequent queries changes very slowly.

On the other hand, posting lists caching promises higher cache hit ratios than results caching, however it does not avoid query evaluation and query evaluation costs may be computationally expensive especially in best-match retrieval models. Moreover, static caching of terms and their corresponding posting lists can be more effective than dynamic caching.

Recent works show that in order to achieve optimal performance a cache should be divided into two parts, a results cache and a posting lists cache.

In general, we could say that the selection of the "right" caching approach depends on (a) the size of the entries to be cached, (b) the tasks whose evaluation we would like to speed up, and (c) the usage scenario at hand. We could distinguish the following general approaches:

- Analyze the frequencies of the tasks and their evaluation costs and from this analysis design a caching mechanism. We could say that results caching falls into this category.

- Break down tasks to subtasks whose evaluation can be cached and then exploit these cached results by more tasks. We could say that posting lists caching falls into this category (as well as approaches that store frequently occurring pairs of terms and the intersection of their corresponding posting lists).
- Evaluate tasks by exploiting the cached results of "similar" tasks, if the extra cost required is less than the cost of performing the task from scratch. In this work we describe such approaches.

Chapter 3

Set Cover Results Caching (SCRC)

In this chapter we introduce and describe in detail the Set Cover-based Results Cache.

The core idea of the SCRC, is to exploit cached sub-queries for answering the incoming queries, aiming at maximizing the utilization of the cache. For example, if we have cached the answers of the queries q_1="barack obama", q_2="facebook", q_3="popularity", then we answer from the cache queries of the form: "barack obama popularity", "barack obama facebook", and "barack obama facebook popularity", and we can also speedup the evaluation of queries like "barack obama inauguration". We reduce the problem of finding the "best" cached sub-queries that are required for answering the incoming query, to the well know *Set Cover Problem* and we adapt the corresponding algorithms to our needs.

We can say that SCRC bridges the gap between results caching and posting lists caching: 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 (which can be considered as a special case of our approach). 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, like Okapi BM25.

Section 3.1 introduces notations. In Section 3.2 we introduce the notion of decomposable scoring functions which determines the applicability of SCRC and we show that several best-match retrieval models (i.e VSM, Okapi BM25 and hybrid retrieval models) rely on such scoring functions. Section 3.3 discusses the set cover problem, and Section 3.4 explains

how the set cover problem relates to the SCRC-based query evaluation and shows that decomposability guarantees correctness. Subsequently, Section 3.5 focuses on the algorithmic perspective of the SCRC-based query evaluation. Finally, Section 3.6 describes a variation of the SCRC method where only the top-K answers are stored in the cache and introduces metrics for measuring the accuracy of answers derived through set cover hits.

3.1 Notations

Let us first introduce some notations. The answer of a query q, denoted by $Ans(q)$, is defined as:

$$Ans(q) = \{\ d \in Obj\ \ |\ \ Score(d, q) > 0\}$$

where $Score(d, q)$ quantifies the relevance of d to q, or/and the importance of d (i.e. PageRank score). Table 3.1 introduces the notations that we use in the sequel.

3.2 Decomposable Scoring Functions

Here we introduce the notion of *decomposable scoring functions* which determines the applicability of SCRC and we show that several best-match retrieval models (i.e VSM, Okapi BM25 and hybrid retrieval models) rely on such scoring functions.

Def 3.2.1 (Decomposability)
Given a query $q = \{t_0, t_1, \ldots, t_n\}$ of n terms, a scoring function $Score(d, q)$ 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.:

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

□

Given a universe of elements $U = \{u_1, u_2, \ldots, u_n\}$ and a family $C = \{S_1, \ldots, S_k\}$ of subsets of U, C is *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

$$Score(d, q) = \sum_{q_c \in C} Score(d, q_c) \tag{3.1}$$

Symbol	**Meaning**
Q_e	a bag of queries
$Set(Q_e)$	the *set* of the distinct queries of Q_e
T	the vocabulary of the inverted index, $T = \{t_0, t_1, \ldots, t_n\}$
Obj	the set of all potential document identifiers (docIds)
N	the total number of documents in the collection, i.e. $\|Obj\|$
I	the inverted index, $I = \{I(t_0), I(t_1), \ldots, I(t_n)\}$
$I(t_i)$	the inverted list of term $t_i \in T$
$tf_{d,t}$	the frequency of term t in document d
$df(t)$	the document frequency of term t
$idf(t)$	the inverse document frequency of term t
$w_{d,t}$	the weight of term t in document d
$w_{q,t}$	the weight of term t in query q
$W(d)$	the norm of the document vector of d
$W(q)$	the norm of the query vector of q
q	a query submitted to the search engine, $q \in Q_e$
$t(d)$	the set of the distinct words of the document d
$t(q)$	the set of the distinct words of the query q
$Score(d, q)$	the score of document d w.r.t query q
$g(d)$	the static quality score of document d, which is independent from any query $q \in Q$
$pr(d)$	the pagerank score of document d
$Sim(d, q)$	the similarity score of document d with respect to q
$Sim_{cos}(d, q)$	the cosine of the angle of the document and query vectors (VSM score)
$Sim_{BM25}(d, q)$	the score of the document d assigned by the Okapi BM25 retrieval model w.r.t q
$Ans(q)$	the answer of the cache for the query q
Cache Notations	
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

Table 3.1: Notations

i.e. the score of a document d is the sum of the scores of d in the answers of the queries in C. As it will be clarified later on, we are mainly interested in retrieval models which are based on such scoring functions, because no post-processing is required for evaluating queries through *set cover* hits.

3.2.1 Examples of Decomposable Scoring Functions

This section shows that several best-match retrieval models rely on decomposable scoring functions.

TF-IDF Weighting schemes: Extensive experience in information retrieval research over many years has clearly demonstrated that the optimal weighting comes from the use of $tf \cdot idf$ weighting schemes. Several $tf \cdot idf$ based scoring models widely used in the literature (i.e in [45, 15]), use decomposable scoring functions for ranking the matching documents of a query q.

The typical scoring function of the $tf \cdot idf$ weighting scheme is

$$Score(d, q) \quad = \quad \sum_{t \in t(d) \cap t(q)} tf_{d,t} \cdot idf(t)$$

This scoring function is *decomposable*, since it is trivial that it satisfies Def. 3.2.1.

In practice, modern WSEs use more sophisticated scoring functions which consider also the use of document normalization factors and other text statistics. Document length normalization is significant for effective retrieval, since long documents have usually a much larger term set than short documents, which makes long documents more likely to be retrieved than short documents.

The Okapi BM25 model and the Vector Space Model have been the most successful term weighing ranking functions in Information Retrieval. Next, we show that their scoring functions are *decomposable*.

The Okapi BM25 Model: The Okapi BM25 model [41] which is based on earlier work on probabilistic inferencing in Information Retrieval (IR) [40] uses the following ranking 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,
- 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 $N = |Obj|$. Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

The scoring function of Okapi BM25 is *decomposable*, since Def. 3.2.1 holds. It holds 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: In contrast, the typical scoring function of the Vector Space Model (VSM) is not decomposable. In this model, both documents and queries are represented as vectors and the similarity measure is derived from the geometrical relationship (cosine of angle) of these vectors.

Each document d is represented by a weighted vector of keywords extracted from the document, where each weight represents the importance of a keyword in the document and within the whole document collection, specifically $w_{d,t} = tf_{d,t} * idf(t)$, i.e:

$$\overrightarrow{d} = \{w_{d,t_1}, w_{d,t_2}, \ldots, w_{d,t_n}\}$$

Each query q is represented also by a vector with associated weights representing the importance of the keywords in the query, i.e:

$$\overrightarrow{q} = \{w_{d,t_1}, w_{d,t_2}, \ldots, w_{d,t_n}\}$$

The relevance between a document d and a query q equals the cosine of the angle of their vectors, denoted by $Sim_{cos}(d, q)$, and is computed by:

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

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. 3.2.1 does not hold. The score of a query q w.r.t to a document d is not equal to the sum of its individual term relevances. The reason is the presence of the W_q term in the denominator, which is query-dependent. This means that in the general case:

$$Sim_{cos}(d, q) \neq \sum_{t \in t(q)} Sim_{cos}(d, t)$$

The proof is given at the Appendix (A.1).

However, the term W_q can be neglected since it does not affect the relative ordering of the documents, nor the retrieval effectiveness [49]. Let $Sim'_{cos}(d, q)$ denote the score assigned to a document d w.r.t q according to the modified version.

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

This variation results in a scoring function which is *decomposable*, because the denominator does not depend on any query term. The proof is trivial:

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

Let us now consider the case, where we do not want to ignore the query norm W_q. This scoring function is what we call *semi-decomposable*. We call a scoring function *semi-decomposable*, if it holds:

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

where X is a parameter set containing d and/or q and $f(X)$ is a function over its elements (d or q), and $\otimes$ stands for one or more arithmetic operations. Such functions require a post-processing for determining the final score of a document. For obtaining the value of $f(X)$ we may have to fetch information from the index, if X includes d. Otherwise, the index does not need to be consulted.

Returning to the VSM, $X = q$, $\otimes = \cdot$ and $f(q) = \frac{1}{W_q}$. Hence the typical scoring function of the Vector Space Model can be expressed as:

$$Sim_{cos}(d,q) = \frac{1}{W_q} \cdot \sum_{t \in t(d) \cap t(q)} (\frac{w_{d,t} \cdot w_{q,t}}{W_d})$$

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

Hybrid scoring functions usually have the form of a linear function of the form:

$$Score(d,q) = a * g(d) + (1-a) * Sim(d,q) \tag{3.2}$$

where $g(d)$ denotes the query-independent score of d and $0 \leq a \leq 1$. 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. It follows from the form of Eq. (3.2), that if $Sim()$ is decomposable then $Score(d,q)$ is semi-decomposable. The proof is trivial since:

$$\begin{aligned} Score(d,q) &= a * g(d) + (1-a) * Sim(d,q) \\ &\overset{\text{(Def.3.2.1)}}{=} a * g(d) + (1-a) * \sum_{t \in t(q)} Sim(d,t) \end{aligned}$$

The resulting scoring function is consistent with the definition of semi-decomposable scoring functions with $X = \{d\}$ and $f(d) = g(d)$.

If $Sim()$ is semi-decomposable then $Score(d,q)$ is also semi-decomposable, but the post-processing operands and operations are more. For example, if $Sim()$ stands for the typical scoring function of the VSM (without discarding W_d), then $X = \{d,q\}$, $f(d) = g(d)$ and $f(q) = \frac{1}{W_q}$. i.e,

$$Score(d,q) = a * g(d) + (1-a) * \left[\frac{1}{W_q} * \sum_{t \in t(d) \cap t(q)} \left(\frac{w_{d,t} \cdot w_{q,t}}{W_d} \right) \right]$$

However, 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*, if the query-dependent score is decomposable. For instance, in [44], the relevance score of a document d w.r.t to query q is computed by the following formula:

$$Score(d,q) = \sum_{t \in t(q)} (bm25(d,t) + w \frac{pr(d)}{pr(d)+k})$$

where $bm25(t,d)$ is the non-normalized BM25 score of the document d for a term t, $pr(d)$ is the query independent score of the document d and w, k are constants that determine the weight of $pr(d)$. This is *decomposable* because the score of the document equals the sum of

the scores of its individual query terms.

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

and this holds because (as in Okapi BM25) each addend corresponding to one query term, does not contain any quantity that depends on a different query term.

In the context of the Mitos WSE, 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:

$$Score_{Hybrid}(d,q) = Sim(d,q) + freq(d,q) \tag{3.3}$$

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 or equal than $|t(q)|$.

If $|t(d) \cap t(q)| < |t(d') \cap t(q)|$, then $Score_{Hybrid}(d,q) < Score_{Hybrid}(d',q)$, since $Sim(d,q) \in [0,1]$.

If $Sim()$ is decomposable, then this scoring function is also decomposable since Def. 3.2.1 holds. The proof is trivial:

$$\begin{aligned} Score_{Hybrid}(d,q) &= Sim(d,q) + freq(d,q) \\ &\overset{\text{(Def.3.2.1)}}{=} \sum_{t \in t(q)} Sim(d,t) + \sum_{t \in t(q)} freq(d,t) \\ &= \sum_{t \in t(q)} [Sim(d,t) + freq(d,t)] \\ &= \sum_{t \in t(q)} Score_{Hybrid}(d,t) \end{aligned}$$

If $Sim()$ is semi-decomposable, then this hybrid scoring is also semi-decomposable.

3.3 Set Cover Problem

This section describes the set cover problem and some variations which are useful for the problem at hand.

Def 3.3.1 (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, a set cover is the minimum in size subfamily C of F such that $\bigcup_{S_i \in C} S_i = U$.

The set cover problem is one of the oldest and most studied problems in the area of combinatorial optimization. However note that a set cover is not necessarily a *partition*. As it will be clarified later on, we are interested in finding *exact set covers*, where an exact cover is a family of subsets of U that cover U however there is the extra constraint that all sets of C should be pairwise disjoint. An example demonstrating the difference between a plain set cover and an exact set cover follows (Ex. 3.3.1).

Example 3.3.1 Figure 3.1 shows an instance of the set covering problem.

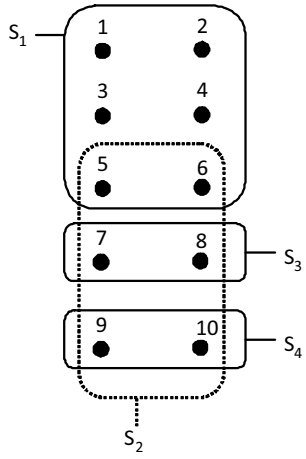


Figure 3.1: Plain Set Cover Vs Exact Set Cover

In this example $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and the family of subsets of U is $F = \{S_1, S_2, S_3, S_4\}$. The set cover in this example is $C = \{S_1, S_2\}$, since

$$\begin{aligned} \bigcup_{S_i \in C} S_i &= S_1 \cup S_2 \\ &= \{1, 2, 3, 4, 5, 6\} \cup \{5, 6, 7, 8, 9, 10\} \\ &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \\ &= U \end{aligned}$$

and $|U|$ is minimal. However, C is not an exact set cover, since sets S_1 and S_2 are not disjoint.

An exact set cover is $C' = \{S_1, S_3, S_4\}$, since

$$\begin{aligned}\bigcup_{S_i \in C'} S_i &= S_1 \cup S_3 \cup S_4 \\ &= \{1,2,3,4,5,6\} \cup \{7,8\} \cup \{9,10\} \\ &= \{1,2,3,4,5,6,7,8,9,10\} \\ &= U\end{aligned}$$

and $S_1 \cap S_3 = \emptyset$, $S_1 \cap S_4 = \emptyset$, and $S_3 \cap S_4 = \emptyset$. $\diamond$

Def 3.3.2 (Exact Set Cover)
We shall say that a family of sets $C = \{S_1, \ldots, S_k\}$ is an *exact set cover*, for short *ESC*, if $\cup_{S_i \in C} = U$ and $S_i \cap S_j = \emptyset$ for every $i \neq j$.

Note that the formulation of the exact set cover problem does not have any kind of minimality constraint. It is actually a decision problem (to find an exact cover or else determine none exists) and it is NP-complete [28].

We shall now introduce the notion of a *partial exact set cover*.

Def 3.3.3 (Partial Exact Set Cover)
We shall say that a family of sets $C = \{S_1, \ldots, S_k\}$ is a *partial exact set cover*, for short *PESC*, if $\cup_{S_i \in C} \subset U$ and $S_i \cap S_j = \emptyset$ for every $i \neq j$.

Obviously a *PESC* is neither a set cover, nor an exact set cover.

3.4 Decomposability, Exact Set Covers and SCRC

In this section we discuss how SCRC is related to the set cover problem. Let Q_c denote the queries whose answers have been cached, and let q be the incoming query. Let first define what we call *lower* queries.

Def 3.4.1 (Lower Query) A query q_c is a *lower* query of q if $t(q_c) \subset t(q)$. We can define the set of cached lower queries as $Lower(q) = \{q_c \in Q_c \mid t(q_c) \subset t(q)\}$

Let $q = \{t_0, t_1, \ldots, t_n\}$ be the new arrived query. If the cache contains an *identical* query q_c, i.e. a query such that $t(q_c) = t(q)$, then $Ans(q) = Ans_{cache}(q_c)$. We use a canonical

representation of queries, i.e. words sorted lexicographically, in order to be invariant to the order of words. If not, we can try to exploit the "closest" to q sub-queries that are cached.

The set cover problem or the exact set cover problem as described in the previous section is related to our problem as follows:

- U corresponds to the terms of q, i.e. $U = t(q)$, while
- F consists of the *lower* cached queries, i.e. $F = Lower(q)$.

Let C be an exact set cover of $t(q)$ corresponding to queries stored in the cache. We shall now define $Ans_C(q)$ and $Score_C(d)$:

$$\begin{aligned} Ans_C(q) &= \cup_{q_c \in C} Ans(q_c) \\ Score_C(d,q) &= \sum_{q_c \in C} Score(d, q_c) \end{aligned}$$

i.e. $Ans_C(q)$ is the union of the documents in the answers of the queries in C, while for a document $d \in Ans_C(q)$ the quantity $Score_C(d,q)$ is the sum of the scores that d received in the answers of C.

This is the core idea of the SCRC query evaluation method. To prove its correctness we need to prove that with this method (a) we will obtain the correct documents, and (b) each of these documents will have the correct score. Formally, we have to prove that (a) $Ans(q) = Ans_C(q)$, and (b) for each $d \in Ans(q)$ it holds $Score(d,q) = Score_C(d,q)$.

If the scoring function $Score()$ is decomposable then (b) certainly holds. Let's now prove (a), i.e that we will get the same set of objects.

Def 3.4.2 We say that a retrieval model is *term-monotonic* if it satisfies the following property: if $t(q) \subseteq t(q') \Rightarrow Ans(q) \subseteq Ans(q')$.

Recall that $Ans(q) = \{\ d \in Obj\ \mid\ Score(d,q) > 0\}$ and note that here we treat answers as sets (we ignore the ranking of its elements).

Lemma 3.4.1 If a scoring function assigns a positive score to a document if it contains at least one query term, i.e. if $t(d) \cap t(q) \neq \emptyset\ \Rightarrow\ Score(d,q) > 0$, then the scoring function is *term-monotonic*.

It is not hard to see that the Vector Space Model as well as the Okapi BM25 model are *term-monotonic*. In general all widely used best match retrieval models are based on term-monotonic scoring functions.

Returning to the problem at hand, term monotonicity guarantees that $Ans(q) = Ans_C(q_c)$ since $t(q) = \cup_{q_c \in C} t(q_c)$.

Lemma 3.4.2 If C and C' are two exact set covers of $t(q)$ and the scoring function is term-monotonic and decomposable, then $Ans_C(q) = Ans_{C'}(q)$ and for each $d \in Ans(q)$ it holds $Score_C(d, q) = Score_{C'}(d, q)$.

Next, we provide an example (Example 3.4.1).

Example 3.4.1 Consider the following simple document collection:

$$d_1 = \text{“}barack\ obama\text{”}, d_2 = \text{“}nobel\text{”}, d_3 = \text{“}nobel\ prize\text{”}, d_4 = \text{“}prize\text{”}$$

and the following cache.

Q_c	**Cached Answer**
q_{c_1}=“barack obama”	$Ans_{cache}(q_{c_1}) = \{(d_2, 2.0)\}$
q_{c_2}=“nobel”	$Ans_{cache}(q_{c_2}) = \{(d_1, 1.0), (d_3, 1.0)\}$
q_{c_3}=“nobel prize”	$Ans_{cache}(q_{c_3}) = \{(d_3, 2.0), (d_1, 1.0), (d_4, 1.0)\}$
q_{c_4}=“prize”	$Ans_{cache}(q_{c_4}) = \{(d_3, 1.0), (d_4, 1.0)\}$

Assume that the query q= “barack obama nobel prize” is submitted to the WSE and for simplicity consider a *decomposable* scoring function assigning scores to the documents via the following formula:

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

where $freq(t, d)$ is the frequency of term t in document d.

In this example, we see that there are 2 exact set covers: $C_1 = \{q_{c_1}, q_{c_3}\}$ and $C_2 = \{q_{c_1}, q_{c_2}, q_{c_4}\}$.

By using C_1 we obtain the following answer:

$$Ans_{C_1}(q) = \bigcup_{q_c \in C_1} Ans_{cache}(q_c) = \{d_2, d_3, d_4, d_1\}.$$

The scores of the documents are: $Score_{C_1}(d_2) = 2.0, Score_{C_1}(d_3) = 2.0, Score_{C_1}(d_4) = 1.0, Score_{C_1}(d_1) = 1.0$.

Hence, $Ans_{C_1}(q) = \{(d_2, 2.0), (d_3, 2.0), (d_4, 1.0), (d_1, 1.0)\}$.

By using C_2 we obtain the following answer:

$$Ans_{C_2}(q) = \bigcup_{q_c \in C_2} Ans_{cache}(q_c) = \{d_2, d_3, d_4, d_1\}$$

The scores of the documents are: $Score_{C_2}(d_2) = 2.0, Score_{C_2}(d_3) = 1.0+1.0 = 2.0, Score_{C_2}(d_4) = 1.0, Score_{C_2}(d_1) = 1.0$.

Hence, $Ans_{C_2}(q) = \{(d_2, 2.0), (d_3, 2.0), (d_4, 1.0), (d_1, 1.0)\}$, which is the same as $Ans_{C_1}(q)$.

◇

Finally, we have to note that if documents are globally ranked by a query-independent score (i.e Pagerank score) then a plain set cover (not necessarily a partition) is sufficient for the correct computation of the final scores of the documents.

3.5 Algorithms

3.5.1 Finding Cached SubQueries

Regarding the cost for finding the *lower* queries of the incoming query we can identify two 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 tested experimentally the benefits of these approaches using a trace of queries submitted to the Excite WSE, where $|Q_e| = 37,096$. Then we classified all the queries according to their length.

We created a cache holding 10^4 queries (i.e. $|Q_c| = 10^4$) and we examined the efficiency of the above approaches by investigating the *lower* queries of each query. Each approach

was tested against queries composed of X or more terms, $1 \leq X \leq 10$. Table 3.2 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.

	$t(q) \geq X$									
	$X = 1$	$X = 2$	$X = 3$	$X = 4$	$X = 5$	$X = 6$	$X = 7$	$X = 8$	$X = 9$	$X = 10$
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

Table 3.2: Strategies for finding lower queries

Firstly, we observe that the hash-based approach substantially outperforms in all cases the scan-based approach. The pure efficiency of the scan-based approach relies on the fact that it always has to scan all the cached queries. This is also demonstrated in Algorithm 2. Regarding the hash-based approach we observe that its execution time increases as the number of the query terms increases but in practise, $|t(q)|$ is usually less than 5.

3.5.2 Finding Exact Set Covers

Set-covering is an NP-Hard problem to solve. A greedy set cover algorithm which achieves logarithmic approximation solution is described in [19]. At the appendix of this thesis we provide this algorithm along with an example (A.2). Below we introduce a variation of the algorithm presented in [19] for finding exact covers. The greedy algorithm chooses sets according to one rule: at each stage choose the set which contains the largest number of uncovered elements. Repeat until all elements are covered. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. The greedy algorithm, called `EfficientGreedyExactSetCover`, is presented next (Algorithm 1).

Running Time. For the problem at hand, Alg. 1 is called as `EfficientGreedyExactSetCover`($\mathtt{Q_c}, \mathtt{q}$). Let's start from the time required to produce the *lower queries* of the incoming query q. This can be done using Alg. `Lower`($\mathtt{Q_c}, \mathtt{q}$) (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.

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. `Lower(Qc, 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 at each iteration $F.removeMax()$ is invoked 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 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 $|F| \leq 2^{|t(q)|} - 2$.

Algorithm 1 returns either an *exact* set cover (ESC) or a *partial* set cover ($PESC$) (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 in order 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. It follows that a $PESC$ C can be extended to become an exact set cover. To the problem at hand, if C is a $PESC$ and $Cterms = \cup_{S_i \in C}$, then we shall use $Rem(q)$ to denote the remainder terms of q, where $Rem(q) = t(q) \setminus Cterms$, where $Cterms$ denotes the terms of the returned cached queries. Clearly, if C is an *exact* set cover, then $Rem(q) = \emptyset$. This means 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 missing terms in $Rem(q)$.

In the sequel we shall use the abbreviations $q = ESC(qset)$, $qset \subset Q_e \setminus \{q\}$ to denote that $qset$ is an *exact* set cover of q found by the greedy algorithm and $q = PESC(qset)$, to

Algorithm 1 `EfficientGreedyExactSetCover(Qc, q)`

```
1: C ← ∅;  U ← t(q); Visited ← ∅
2: F ← Lower(Qc, q)
3: while U ≠ ∅ do
4:     S ← F.removeMax()                          ▷ Top-node of the heap
5:     // If S cannot lead to an exact set cover:
6:     if ((S ≠ null) and (S ∩ Visited ≠ ∅)) then
7:         continue                               ▷ Skip this iteration
8:     end if
9:     if (S = null) then                         ▷ The heap is empty
10:        if (C = ∅) then
11:            return (null, NOTHING)             ▷ Nothing found
12:        else
13:            return (C, PESC)                   ▷ Partial exact set cover found
14:        end if
15:    else
16:        C ← C ∪ {S}
17:        U ← U − S
18:        Visited ← Visited ∪ S
19:    end if
20: end while
21: return (C, ESC)                               ▷ Exact set cover found
```

Algorithm 2 `Lower(Qc, q)`

```
1: F ← ∅;                                         ▷ F is a max heap
2: if ( |t(q)| < log(|Qc| + 2) ) then             ▷ Hash-based lookup
3:     SubQueries ← P(q) \ {t(q), {}}             ▷ P(q): Powerset of q
4:     for (each Pi ∈ SubQueries) do
5:         if (Pi ∈ Qc) then                      ▷ Access through Hash
6:             F.insert(|Pi|, Pi)                 ▷ Insert Pi in the heap
7:         end if
8:     end for
9: else                                           ▷ Scan-based approach
10:    for each qc ∈ Qc do
11:        if (qc ⊂ q) then
12:            F.insert(|t(qc)|, qc)              ▷ Insert qc in the heap
13:        end if
14:    end for
15: end if
16: return F
```

denote that *qset* is a *partial* exact set cover of q found by the greedy algorithm. In the latter case, we consider $PESC(qset)$ to be a *partial* set cover, if $\nexists\ qset'$ such that $q = ESC(qset')$, $qset' \subset Q_e \setminus \{q\}$.

3.5.3 Cache Structure

Regarding the cache structure, the cache stores the document id and the relevance score for each document in a cached answer. There is no need for storing anything more. Table 3.3 shows the cache structure, which is the same as the structure of a RC.

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}	$Ans_{cache}(q_{c_n}) = \{(d_i, Score(d_i, q_{c_n})), \ldots\}$

Table 3.3: SCRC Cache Structure

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.

3.5.4 Query Evaluation

Now we describe how we exploit the cache for answering the incoming query. The basic idea of SCRC is the following: if there is not an identical query in the cache, check if the terms of the cached queries can *cover* (totally or partially) the terms of the incoming query. In case of a partial exact set cover fetch the missing posting lists from the index.

Let q be the incoming query, and let C be the family of sets returned by `EfficientGreedyExactSetCover(q,Qc)`. If C is an exact set cover (ESC), then the answer is provided by the cache. If C is a partial exact set cover ($PESC$) 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.

Algorithm 3 `getAnswer(Qc, q)`

if $(Q_c.get(q) \neq null)$ **then** ▷ identical hit
  return $Q_c.get(q)$
else
  $(C, note) \leftarrow$ `EfficientGreedyExactSetCover(q, Qc)`
  if $(note = null)$ **then**
    return null ▷ should be processed from the main index
  end if
  if $(note = ESC)$ **then**
    $Ans_C(q) \leftarrow \bigcup_{q_c \in C} Ans_{cache}(q_c)$
  end if
  if $(note = PESC)$ **then**
    $PartialAnswer \leftarrow \bigcup_{q_c \in C} Ans_{cache}(q_c)$
    $CTerms \leftarrow \bigcup_{q_c \in C} t(q_c)$
    $Rem(q) \leftarrow t(q) \setminus CTerms$ ▷ remainder of q
    $C \leftarrow C \cup \{Rem(q)\}$
    $Ans_C(q) \leftarrow Union(PartialAnswer, Ans(Rem(q))$
  end if
  for each $d \in Ans_C(q)$ **do**
    Set $Score_C(d, q) \leftarrow \sum_{q_c \in C} Score(d, q_c)$
  end for
  Sort $Ans_C(q)$ wrt $Score_C(\cdot, q)$
  return $Ans_C(q)$
end if

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 $Score_C(d, q)$, as computed by the cache. Note that the post-processing operations and operands depend on the form of the scoring function, as we described in Section 3.2.

Furthermore, we have to note also that by forcing *AND* semantics in order to consider only those documents containing all the query terms of the incoming query q (as perhaps some commercial WSEs do), this does not affect 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 query terms of q, then d will be included in $Ans_C(q)$, iff d appears in all the cached answers of C. Regarding the query evaluation algorithm, we need only change the union operation for obtaining the matching documents of the incoming query to an intersection operation (lines 9, 12 and 16 of Alg. 3) in order to retain only those documents that appear in all the cached answers. 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 (the answers are sets), while in the second case, we must retain only those documents that appear in all the cached answers.

A second way to consider only those documents containing all the query terms of the incoming query q is through the use of the decomposable hybrid scoring function (Eq. 3.3) described in Section 3.2 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.

3.6 Top -K SCRC

Most of the RC approaches cache top-K results (for K ranging some hundreds). In this section we present an extension of the SCRC for caching the top-K results of a query (for K ranging some hundreds). If in a SCRC we store only the top-K documents of the cached queries, then an "identical hit" will return the correct top-K result. However a *"set cover*

hit" will not necessarily return the correct top-K result.

Consider a submitted query q, let C be a 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. $B = \cup_{q_c \in C} Ans_{cache}(q_c)$. Let's first investigate the two extremes: if a document b appears in the answers of all queries in C, then the final score of b will be the correct one, while 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 suppose that b appears in some of the answers of the queries in C. That document will get a score smaller than what it deserves if a best match retrieval model is employed (in query independent ranking models its score will be correct), and this happens if the position of b at the full answer of a cached query in C, say q_c, is the position $k+1$ or a greater position. However we can compute an upper bound of the score that b can have by assuming that in those answers where b is absent, b could take at most the score of the last element. 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' ($C' \subset C$) is 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

$$score_{certain}(b) \leq score(b) \leq score_{up}(b)$$

We can exploit the upper bounds of the missed scores as well as 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 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 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. Clearly it holds always that $K_{ro} \leq |C| * K$. For the computation of K_{ex} we have to scan all elements of B which are at most $|C| * K$ if the answers in all set covers are pairwise disjoint. However, the last element of each answer list can never exceed $Missing^{up}$. The count of these lists is $|C|$, thus there are $|C|$ elements that have certain score less than $Missing^{up}$. Thus, the maximum value of K_{ex} is $|B| - |C|$ $= |C| * K - |C| = |C| * (K - 1)$.

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 $Missing^{up} \leq Found^{up}(B)$. The values K_{ex} and K_{ro} can now be computed in a simple and efficient manner. We order the objects in B in descending order with respect to their *certain* scores. 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 the 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 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. Next we prove the correctness of the Top-K SCRC (Prop. 3.6.1) and we provide several examples (Examples 3.6.1, 3.6.2, 3.6.3).

Prop 3.6.1 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 w.r.t 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:

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

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 $Ans(q)_{|\{e_1,e_2,e_3\}}$.

$\diamond$

Example 3.6.1

Assume the following top-4 SCRC

Q_c	*Cached answers*
q_{c_1}	$Ans_{cache}(\text{"}b\ c\text{"}) = \{(d_2, 0.8), (d_1, 0.4), (d_3, 0.3), (d_4, 0.2)\}$
q_{c_2}	$Ans_{cache}(\text{"}a\text{"}) = \{(d_1, 0.6), (d_3, 0.4), (d_2, 0.2), (d_5, 0.1)\}$

and a newly arrived query $q =$ "$a\ b\ c$". 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 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

It follows that

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

By ordering B in descending order w.r.t their certain scores we get:

$$B = \langle (d_1, 1.0), (d_2, 1.0), (d_3, 0.7), (d_4, 0.2), (d_5, 0.1) \rangle$$

To find K_{ro} we proceed as long the following inequalities hold:

$$\begin{aligned} score_{cert}(d_1) = 1.0 &\geq Found^{up}(\{d_2, d_3, d_4, d_5\}) = 1.0 \\ score_{cert}(d_2) = 1.0 &\geq Found^{up}(\{d_3, d_4, d_5\}) = 0.7 \\ score_{cert}(d_3) = 0.7 &\geq Found^{up}(\{d_4, d_5\}) = 0.3 \\ score_{cert}(d_4) = 0.2 &\geq Found^{up}(\{d_5\}) = 0.3 \\ score_{cert}(d_5) = 0.1. &\geq Found^{up}(\emptyset) = 0 \end{aligned}$$

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

$\diamond$

Example 3.6.2

Assume the following top-4 SCRC

Q_c	*Cached answers*
q_{c_1}	$Ans_{cache}(\text{"}b\ c\text{"}) = \{(d_1, 0.9), (d_2, 0.9), (d_3, 0.8), (d_4, 0.1)\}$
q_{c_2}	$Ans_{cache}(\text{"}a\text{"}) = \{(d_5, 0.9), (d_6, 0.8), (d_7, 0.1)\}$

and a newly arrived query q = "*a b c*". There is a "set cover hit", $C = \{q_{c_1}, q_{c_2}\}$, and $B = \cup_{q_c \in C} Ans_{cache}(q_c) = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$. The certain, missed and upper bound scores are shown next:

	score		
d_i	certain	up miss	up
d_1	0.9	0.1	1.0
d_2	0.9	0.1	1.0
d_3	0.8	0.1	0.9
d_4	0.1	0.1	0.2
d_5	0.9	0.1	1.0
d_6	0.8	0.1	0.9
d_7	0.1	0.1	0.2

$$\begin{aligned} Missing^{up} &= 0.1 + 0.1 = 0.2 \text{ (the sum of the last scores)} \\ Found^{up}(B) &= 1.0 \text{ (that of } d_1 \text{ or } d_2 \text{ or } d_5) \end{aligned}$$

By ordering B in descending order w.r.t their certain scores we get:

$$B = \langle (d_1, 0.9), (d_2, 0.9), (d_5, 0.9), (d_3, 0.8), (d_6, 0.8), (d_4, 0.1), (d_7, 0.1) \rangle$$

To find K_{ro} we proceed as long the following inequalities hold:

$$score_{cert}(d_1) = 0.9 \quad \geq \quad Found^{up}(\{d_2, d_5, d_3, d_6, d_4, d_7\}) = 1.0$$

Notice that the 1st inequality does not hold, so $K_{ro} = 0$. K_{ex} equals 5 as the first five documents have score $\geq Missing^{up} = 0.2$. $\diamond$

Example 3.6.3

Assume the following top-4 SCRC

Q_c	*Cached answers*
q_{c_1}	Ans_{cache}("*b c*") $= \{(d_1, 0.9), (d_2, 0.8), (d_3, 0.7), (d_4, 0.1)\}$
q_{c_2}	Ans_{cache}("*a*") $= \{(d_5, 0.6), (d_6, 0.5), (d_7, 0.1)\}$

and a newly arrived query q = "*a b c*". There is a "set cover hit", $C = \{q_{c_1}, q_{c_2}\}$, and $B = \cup_{q_c \in C} Ans_{cache}(q_c) = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$. The certain, missed and upper bound scores are shown next:

	score		
d_i	certain	up miss	up
d_1	0.9	0.1	1.0
d_2	0.8	0.1	0.9
d_3	0.7	0.1	0.8
d_4	0.1	0.1	0.2
d_5	0.6	0.1	0.7
d_6	0.5	0.1	0.6
d_7	0.1	0.1	0.2

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

By ordering B in descending order w.r.t their certain scores we get:

$$B = \langle (d_1, 0.9), (d_2, 0.8), (d_3, 0.7), (d_5, 0.6), (d_6, 0.5), (d_4, 0.1), (d_7, 0.1) \rangle$$

To find K_{ro} we proceed as long the following inequalities hold:

$$\begin{aligned} score_{cert}(d_1) = 0.9 &\geq Found^{up}(\{d_2, d_3, d_5, d_6, d_4, d_7\}) = 0.9 \\ score_{cert}(d_2) = 0.8 &\geq Found^{up}(\{d_3, d_5, d_6, d_4, d_7\}) = 0.8 \\ score_{cert}(d_3) = 0.7 &\geq Found^{up}(\{d_5, d_6, d_4, d_7\}) = 0.7 \\ score_{cert}(d_5) = 0.6 &\geq Found^{up}(\{d_6, d_4, d_7\}) = 0.6 \\ score_{cert}(d_6) = 0.5 &\geq Found^{up}(\{d_4, d_7\}) = 0.2 \\ score_{cert}(d_4) = 0.1 &\geq Found^{up}(\{d_7\}) = 0.2 \\ score_{cert}(d_7) = 0.1 &\geq Found^{up}(\{\emptyset\}) = 0.0 \end{aligned}$$

K_{ro} equals to 6 since the first five inequalities hold. K_{ex} equals 5 as the first five documents have score $\geq Missing^{up} = 0.2$. $\diamond$

Chapter 4

Experimental Evaluation

4.1 Query Log Analysis

Typically, Web Search Engines retain logs of user queries, which register the history of the submitted queries among other data. The analysis of search engine logs has focused on how searchers use the search engines on the Web in order to satisfy their information needs and lies in the research area of the *Web Usage Mining* [46, 18, 4]. Web usage mining is the application of data mining techniques to discover usage patterns from Web data, in order to understand and serve the needs of Web based applications. Some users might be looking at only textual data whereas some others might want to get multimedia data. There has been a large body of work that has been devoted to the analysis of WSE query logs [24, 26, 43, 25, 11] and how users interact with search engines.

In WSEs, query logs constitute a valuable source for designing and evaluating efficient caching policies. Due to the high locality in user queries that logs reveal [34, 47], knowledge about previously submitted queries allows identifying the most frequent queries and cache their results, which is also our focus. Moreover, knowledge about the number of pages of results that a user is interested in, allows defining the optimal number of cached results. Although the query logs of all WSEs do not obey a strict format and each search engine retains various information for each submitted query, typically each request for a query is a record containing:

- the query as it was submitted by the user, including any advanced query operators

- a unique user id (to preserve the anonymity of the IP address of the user), assigned to the user who submitted the query
- the date or timestamp of the submitted query
- the requested page of results (e.g the first page, the second page)

In this context, in order to verify the benefits of set cover-based results caching we conducted an analysis over seven real query logs of three different WSE's. Our aim was to identify the proportion and the characteristics of the queries in a real logs that can be formulated as the disjoint union of the rest queries. For reason of brevity, we will call such queries in the sequel as *set cover queries*.

4.1.1 The Query Logs

Firstly, we describe the query logs that we used in our analysis. More particularly, we analyzed characteristics using query logs of:

- the *Excite* [1] WSE,
- the *Altavista* [2] WSE and,
- the *AllTheWeb* [3] WSE

reported in Table 4.1. All queries contained in these logs were firstly preprocessed by removing empty queries or queries containing any special characters or punctuation marks like phrase queries and urls [4]. For all the remaining queries, we converted all the query terms to lowercase and reordered them in alphabetical order, while stopwords (i.e articles, pronounces) were removed. We also removed requests for further result pages (traditionally considered as duplicate queries). Although such requests are beneficial for WSEs for determining the number of results that the user visited, they skew the results in analyzing how the user searched on system. For instance, it is difficult to discriminate whether a recorded request corresponds to a visit to the second page of results, or to a retyping and resubmission

[1] www.excite.com
[2] www.altavista.com
[3] www.alltheweb.com
[4] For instance, we eliminated queries containing any of the punctuation or special characters: '.', ':', ';', ',', '?', '!', ' ', ')', '[', ']', '{', '}', '"', '/', '\', '+', '-', '*', '#', '&', '@', '%'

of the query. This happens because when a user submits a query, views a document and then returns to the search engine, most of the search engines log this second visit with the same user identification and query, but with a new time (i.e. the time of the second visit). In order to be consistent in the analysis of all the query logs we discarded all the duplicate queries for each user. To identify the duplicate queries submitted by each user we use the *user id* field of each record in the logs.

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

Table 4.1: Query Logs

4.1.2 Query Stream Metrics

To characterize a stream Q_e we introduce four metrics. In their formulas 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:

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

  Clearly, if all queries of Q_e are distinct, then $IQR(Q_e) = 0$. If Q_e consists of one repeated query, then $IQR(Q_e)$ approaches 1 (specifically, $IQR(Q_e) = 1 - \frac{1}{|Q_e|}$).

- To characterize a stream with respect to its density of *partial* exact set covers we introduce the *Partial Set Cover Density (PESCD)* which is the proportion of the queries in Q_e whose terms can be covered partially by the rest queries in Q_e:

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

- 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 exactly by the rest queries in Q_e:

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

The nominator in $PESCD(Q_e)$ and $SCD(Q_e)$ metrics contains the condition $qset \subseteq Set(Q_e) \setminus \{q\}$, because we do not want to count identical hits.

Note that for a query stream Q_e, the sum of the $IQR(Q_e)$, the $PESCD(Q_e)$ and the $SCD(Q_e)$ values does not need to be equal to 1 (100%). For instance, if $Q_e = \{q_1 =$ "$a\ b\ c$", $q_2 =$ "$a\ b$", $q_3 =$ "c", $q_4 =$ "$a\ b\ c$", $q_5 =$ "b"$\}$, then $IQR(Q_e) = 2/5 = 0.4$, $PESCD(Q_e) = 3/5 = 0.6$, and $SCD(Q_e) = 2/5 = 0.4$. Hence, in this case $IQR(Q_e) + PESCD(Q_e) + SCD(Q_e) = 1.4 > 1$.

On the other hand if Q_e contains only distinct queries, then the sum of the values of these query stream metrics cannot exceed 1. It can be at most 1, if all the queries have either an ESC or a $PESC$. For instance, if $Q_e = \{q_1 =$ "$a\ b\ c$", $q_2 =$ "$a\ b$", $q_3 =$ "c"$\}$, then $IQR(Q_e) = 0$, $PESCD(Q_e) = 2/3$, and $SCD = 1/3$. Hence, in this case $IQR(Q_e) + PESCD(Q_e) + SCD(Q_e) = 1$.

4.1.3 Query Length Analysis

Firstly, we measured the length in words of the queries over the logs. Our aim is not to provide a thorough analysis of the users web search behavior, since there are several studies that have analyzed the queries that users submit to search engines as well as the length of search sessions [43, 26, 34]. Instead, we conduct the following analysis in order to compare

the query length level of the original queries with the one of the *set cover* queries, which we describe in the next section.

Figure 4.1 shows the length of the queries over all the used query logs. Regarding the logs of the *Excite* WSE (Figure 4.1(a), (b), (c) and (d)) and the *AllTheWeb* WSE (Figure 4.1 (e) and (f)), we observe that:

- 25% - 30% are single term queries
- about 40% are 2 - term queries
- 20% are 3 - term queries
- 10% consist of more than 3 terms

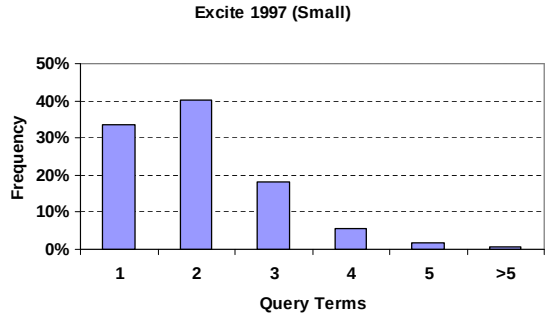
Regarding the Altavista query log (Figure 4.1(g)) we observe that:

- about 20% are single term queries
- 30% are 2 - term queries
- 20% are 3 - term queries
- 30% consist of more than 3 terms

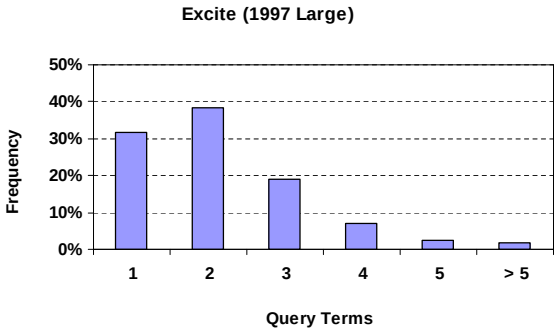
Table 4.2 shows the average query length for each of these logs. On the average, a query contains 2.26 terms, which is consistent with previous studies. This analysis verifies that users use few words in their queries, and shows that the *hash-based* approach that we described in Section 3.5.1 is the most appropriate for exploiting the cached subqueries.

Query Log	Date	$\|Q_e\|$	$AVGQLEN(Q_e)$
Excite	March 13, 1997	36,923	2.03
Excite	Sept. 19, 1997	392,503	2.16
Excite	Dec. 20, 1999	1,125,691	2.25
Excite	May 4, 2001	479,669	2.16
Altavista	Sep. 9, 2002	1,154,598	2.88
AllTheWeb	Feb. 6, 2001	462,678	2.25
AllTheWeb	May. 28, 2002	764,045	2.11

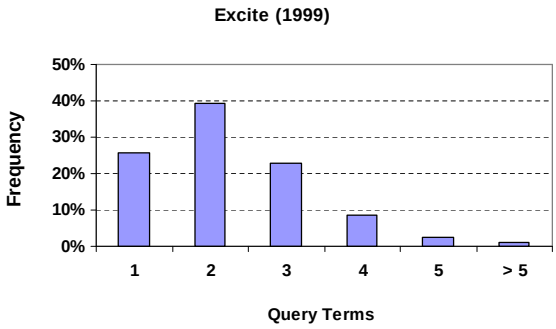
Table 4.2: Average query length of the queries over the logs



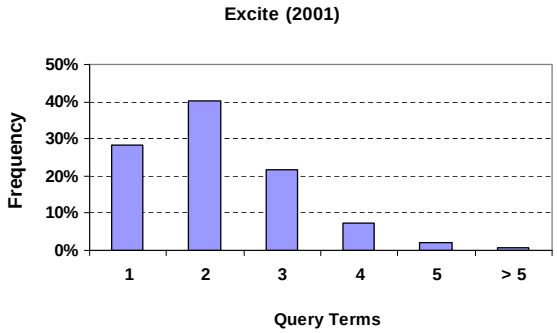
(a) *Excite* (1997-small)



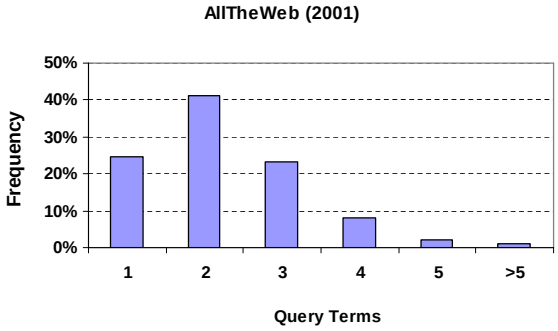
(b) *Excite* (1997-large)



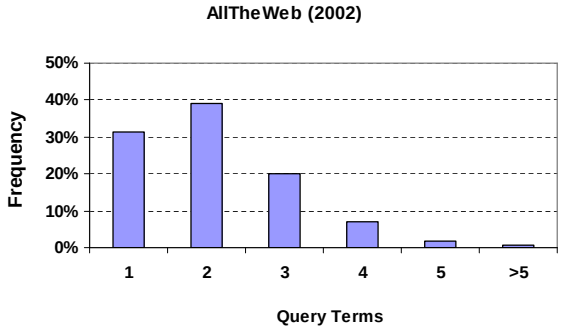
(c)*Excite* (1999)



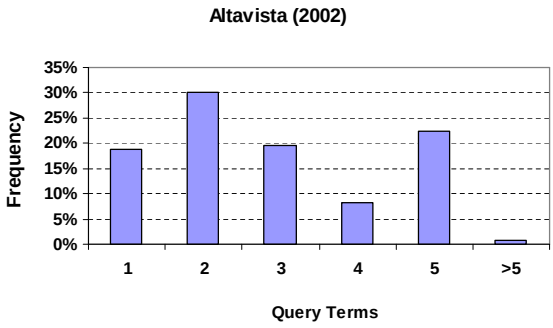
(d) *Excite* (2001)



(e) *AllTheWeb* (2001)



(f) *AllTheWeb* (2002)



(g) *Altavista*

Figure 4.1: Query length in the logs

4.1.4 Analysis of Set Cover Queries

Now we extend our analysis in order to examine the characteristics of the *set cover* queries. For identifying the *set cover* queries over all the logs we used the greedy algorithm for exact set covers (Alg. 1) presented in Section 3.5.2.

In Section 4.1.4.1 we measure the set cover density over the logs. Then, in Section 4.1.4.2, we analyze the length of these queries and in Section 4.1.4.3 we examine the size of the cover of these queries. Finally, in Section 4.1.4.4, we examine the benefits of our approach over these logs, by comparing the cache hit rate of the SCRC and the RC.

4.1.4.1 Set Cover Density

To see what SCD values characterize real query streams, we computed this metric over the logs. Table 4.3 reports the statistics for each of the logs. The $PESC$ and SCD values were obtained by using the greedy algorithm (Alg .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

Table 4.3: Statistics of real query logs using Algorithm 1

The average SCD value over these query logs is 37%, the average $PESCD$ value is 27.5% and the average IQR value is 33%.

Therefore, one out of three queries can be formulated exactly as a combination of the terms of the rest queries and a lower proportion of these queries can be formulated partially by the rest queries. We observe that the average $PESCD$ value over these logs is lower than the average SCD value by 10%. The value of the $PESCD$ is lower, since as we reported we consider a $PESC(qset)$ to be a *partial* set cover, if $\nexists\ qset'$ such that $q = ESC(qset')$, $qset \subset Q_c \setminus \{q\}$. This means that we do not count these queries to the $PESCD$ value, despite the fact that when a query has an exact set cover, it surely has a partial set cover.

The reason that we count them in this way is that in our context either an ESC or a $PESC$ will be returned for a given query.

We also used a non-greedy algorithm for counting the SCD values over these logs, which examines all the combinations of the cached sub-queries of the incoming query. Our objective was to verify that our algorithm does not lead to pure retrieval of exact set covers. The results by using a non-greedy algorithm are presented in Table 4.4. Indeed, a non-greedy algorithm offers at most 0.2%-0.6% more exact set covers than the greedy one.

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%

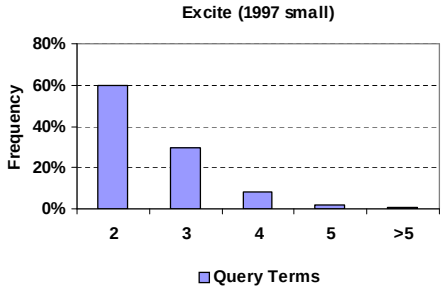
Table 4.4: Statistics of real query logs using a non-greedy algorithm

4.1.4.2 Query Length

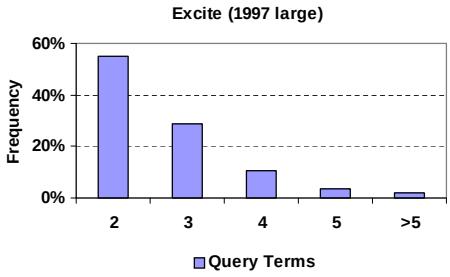
In this section we study the length in words of the set cover queries. We conducted this analysis in order to examine if the characteristics of these queries are close to those of all the queries. Figure 4.2 shows the query length distribution of the *set cover* queries over all the query logs. Firstly, we observe that there are not significant deviations from the analysis that we conducted over all the queries. More precisely, we observe that queries consisting of 2 terms dominate again and there is just a small fraction of set cover queries that are composed of more than 3 terms: We make the following observations:

- 50% - 55% of these queries are 2-term queries
- 30% of these queries are 3-term queries
- 10% - 15% of these queries are composed of more than 3 terms

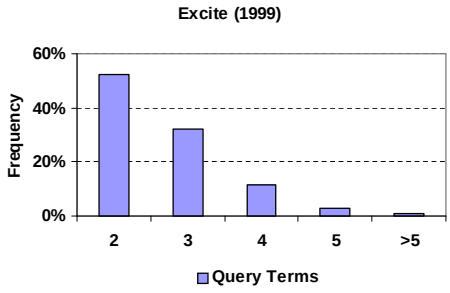
Hence, queries that can be formulated as a disjoint union of the rest queries are also short queries, since the vast majority of these queries have 2 terms.



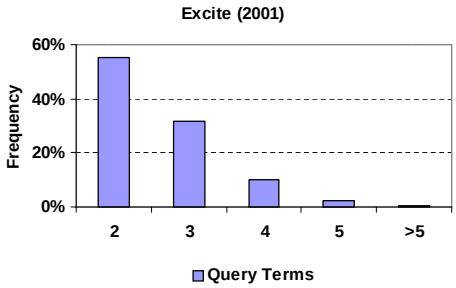
(a) *Excite* (1997-small)



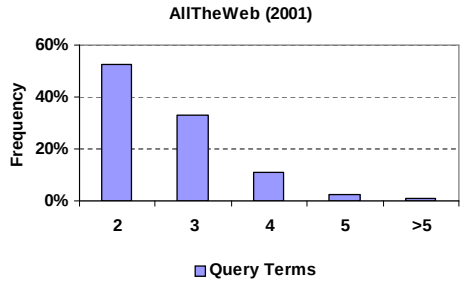
(b) *Excite* (1997-large)



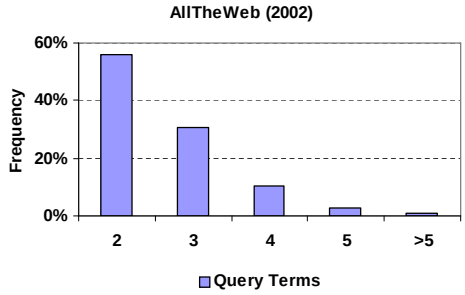
(c)*Excite* (1999)



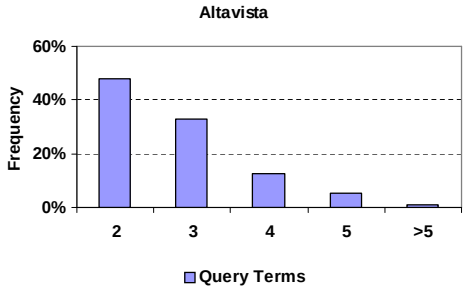
(d) *Excite* (2001)



(e) *AllTheWeb* (2001)



(f) *AllTheWeb* (2002)



(g) *Altavista*

Figure 4.2: Length of the set cover queries

Table 4.5 shows the average length of the *set cover* queries for each of these logs. On the average, a set cover query contains 2.65 terms. We observe that this value is slightly higher than the one obtained for all the queries of the logs. However, this is expected since there cannot be a set cover query that consists of just a single term.

Query Log	Date	$\|Q_e\|$	$AVGQLEN(Q_e)$
Excite	March 13, 1997	36,923	2.52
Excite	Sept. 19, 1997	392,503	2.70
Excite	Dec. 20, 1999	1,125,691	2.68
Excite	May 4, 2001	479,669	2.61
Altavista	Sep. 9, 2002	1,154,598	2.79
AllTheWeb	Feb. 6, 2001	462,678	2.66
AllTheWeb	May. 28, 2002	764,045	2.62

Table 4.5: Average query length of the set cover queries over the logs

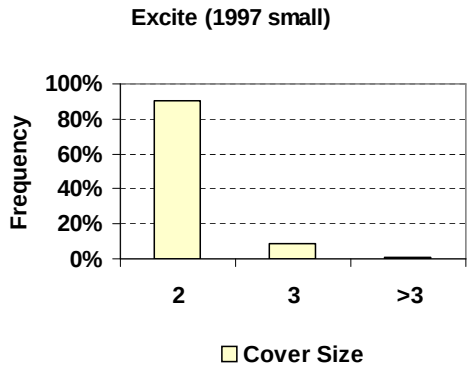
4.1.4.3 Set Cover Size

Now, we focus on the size of the cover of these queries. The cover size of a *set cover* query is the number of the sub-queries which formulate it (which is independent of the number of the query terms that each sub-query contains), e.g if $C = \{\{a\ b\ c\}, \{d\ e, f\}\}$, then the size is 2.

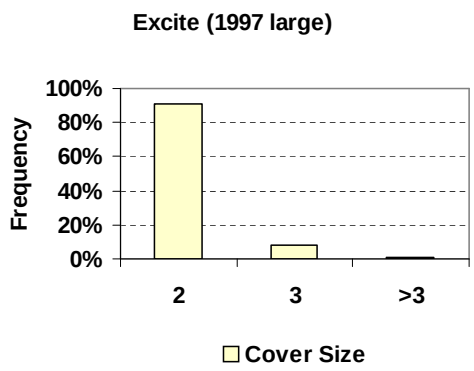
Figure 4.3 shows the cover size of the *set cover* queries over the logs. We observe that the majority of the queries (80% - 85%) have a cover size equal to 2. There is a very small fraction of queries (10% - 13%) with cover size equal to 3 and an even lower fraction (3%) of queries that have a cover size greater than 3 sets. Consequently, we see that the cover size of the set cover queries is small.

4.1.4.4 Cache Hit Rate

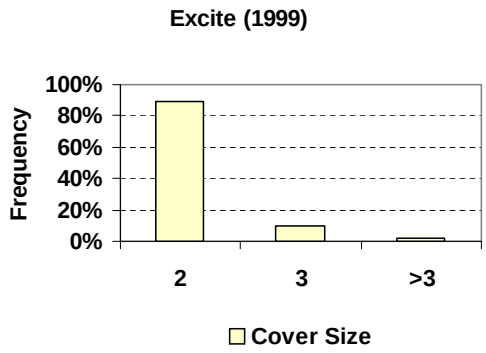
In this section, we compare the hit rate of the RC and the SCRC. The value of the SCD metric in a query stream is a good indicator for revealing how many queries can be covered exactly by the rest queries and indicates the maximum number of set cover hits that a SCRC can achieve. However, it cannot guarantee that all of these queries will be answered though set cover hits.



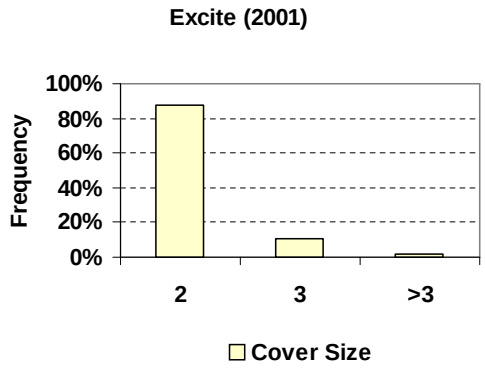
(a) *Excite* (1997-small)



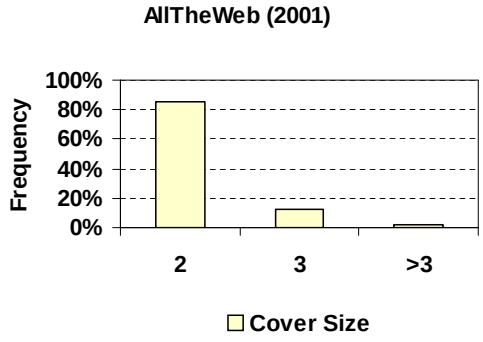
(b) *Excite* (1997-large)



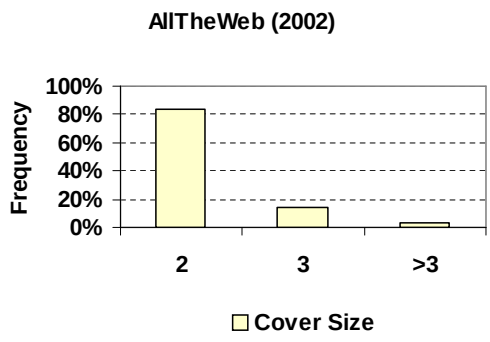
(c)*Excite* (1999)



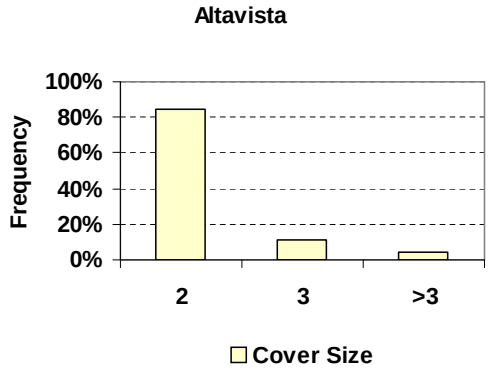
(d) *Excite* (2001)



(e) *AllTheWeb* (2001)



(f) *AllTheWeb* (2002)



(g) *Altavista*

Figure 4.3: Size of the set cover queries over the query logs

- 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 we 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.

Cache Metrics: We refine the metrics defined earlier in Section 4.1.2 in order to count the number of identical 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:

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

The cache hit rate of the RC is equal to the proportion of the identical hits, while the cache hit rate of the SCRC is equal to the proportion of the identical hits and 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 either by the cache through an identical hit or by an exact set cover hit or by using a partial exact set cover, the sum of the values of the cache metrics for a given cache size must always be less than 1.

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 equal size parts: a *training set* and a *test set*, as Markatos [34] [5]. We use the training set to extract the most popular queries and fill the caches and we use the test set to measure the cache hit rate. The contents of both caches are static and do not change during the submission of the queries contained in the test set.

Next, we report our results for various cache sizes over the query logs. For each query log that we use, we plot the cache hit rate of the RC and the SCRC. We also report for each cache size, the values of the cache metrics which reflect the proportion of the identical hits, the proportion of the exact set cover hits and the proportion of the partial exact set covers. Note that we do not count the partial exact set covers in the cache hit rate of the SCRC.

Figure 4.4 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 13.6% - 20.4% higher hit rates than the RC.

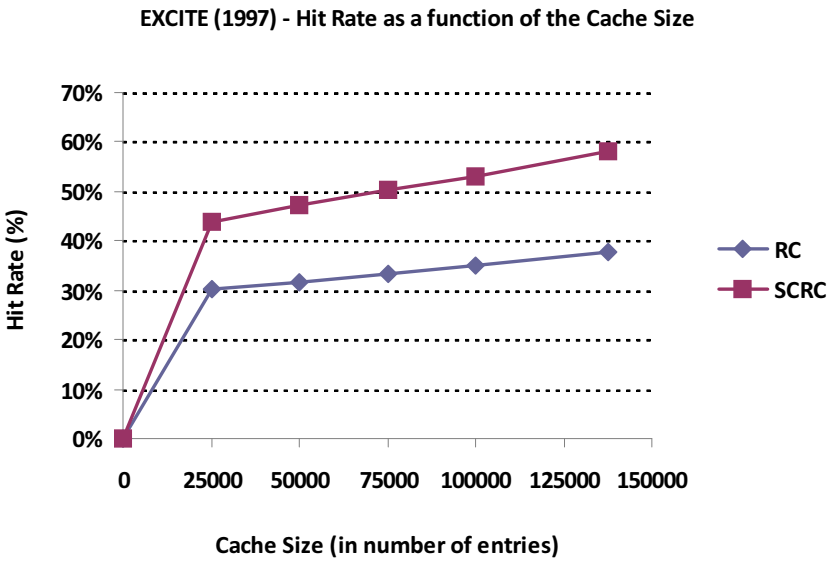


Figure 4.4: Hit Rate of the RC and the SCRC as a function of the Cache Size - Excite (1997) log

Table 4.6 reports in detail the fraction of the identical hits, the exact set covers and the

[5] Fagni et. al [21] follow the same approach as Markatos[34], but they split the query log into a training set that contains the 2/3 of the queries and the rest (1/3) constitute the test set.

partial exact set covers that were obtained for each cache size. We observe that as the cache size increases:

- the identical hits also increase
- the exact set cover hits also increase almost at the same rate as the identical hits
- the partial exact set covers decrease but at a lower rate

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
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 4.6: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the Excite (1997) query log

We continue our analysis using the Excite (1999) query log. Figure 4.5 shows the cache hit rate of the RC and the SCRC over this log. We see that the SCRC attains 22% - 28% higher hit rates than the RC, which is higher than the obtained hit rate in the Excite (1997) log.

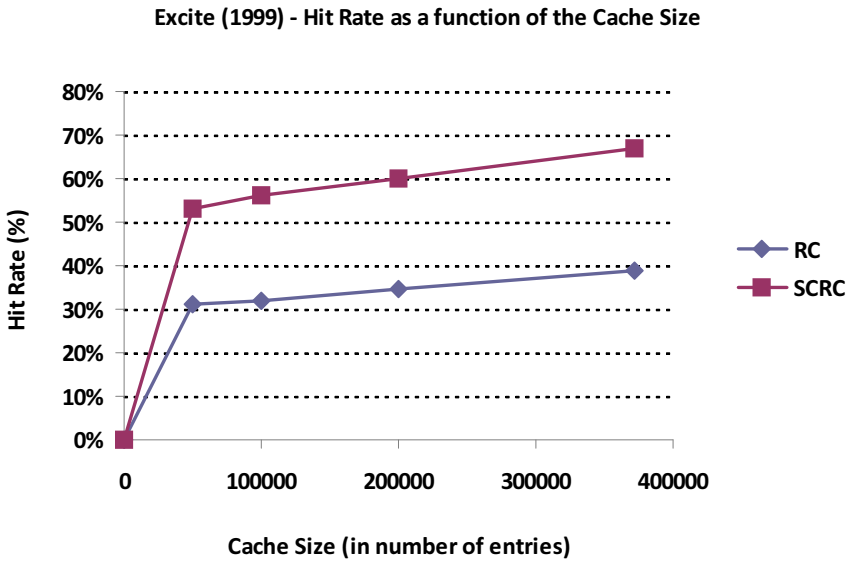


Figure 4.5: Hit Rate of the RC and the SCRC as a function of the Cache Size - Excite (1999) log

Accordingly, Table 4.7 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 over the Excite (1999) log. 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 hardly the same.

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
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 4.7: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the Excite (1999) query log

Figure 4.6 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 15% - 20.5% higher hit rates than the RC.

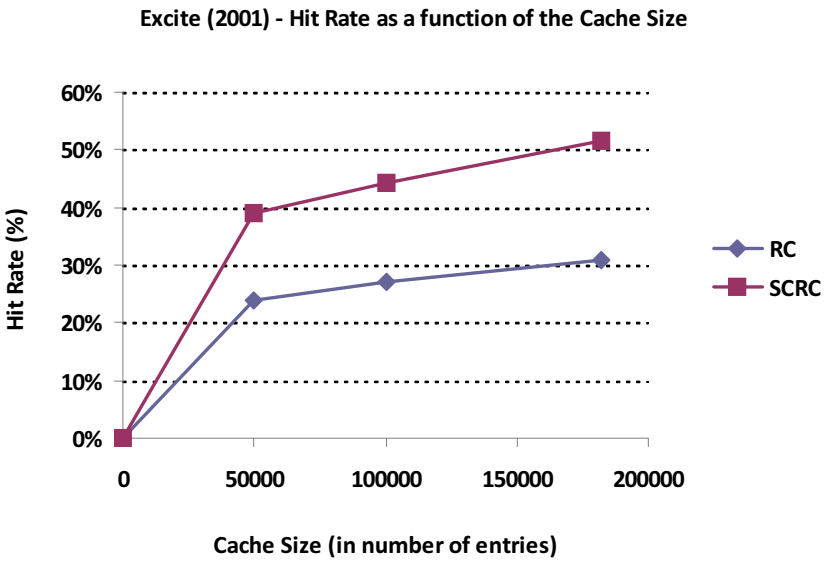


Figure 4.6: Hit Rate of the RC and the SCRC as a function of the Cache Size - Excite (2001) log

Table 4.8 shows the fraction of the identical hits, the exact set covers and the partial exact set covers that were obtained for each cache size over the Excite (2001) log. 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.

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
50,000	24%	15%	33.8%
100,000	27.1%	17.1%	32.4%
181,954	31%	20.5%	29.4%

Table 4.8: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the Excite (2001) query log

Now, we study the hit rate of the RC and the SCRC of the logs of the AllTheWeb WSE.

Figure 4.7 shows the cache hit rate of the RC and the SCRC over the AllTheWeb (2001) query log. We see that the SCRC attains 9.1% - 14.6% higher hit rates than the RC.

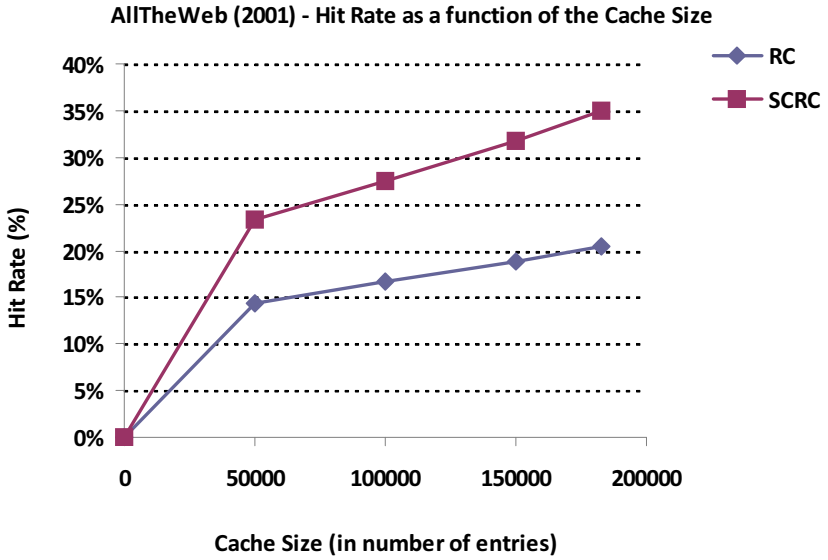


Figure 4.7: Hit Rate of the RC and the SCRC as a function of the Cache Size - AllTheWeb (2001) log

Table 4.9 shows the fraction of the identical hits, the exact set cover hits and the partial exact set covers that were obtained for each cache size over the AllTheWeb (2001) log.

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.

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
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 4.9: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the AllTheWeb (2001) query log

Next, we study the hit rates over the second log of the AllTheWeb WSE. Figure 4.8 shows the cache hit rate of the RC and the SCRC over the AllTheWeb (2002) query log. We see that the SCRC attains 18.8% - 25% higher hit rates than the RC.

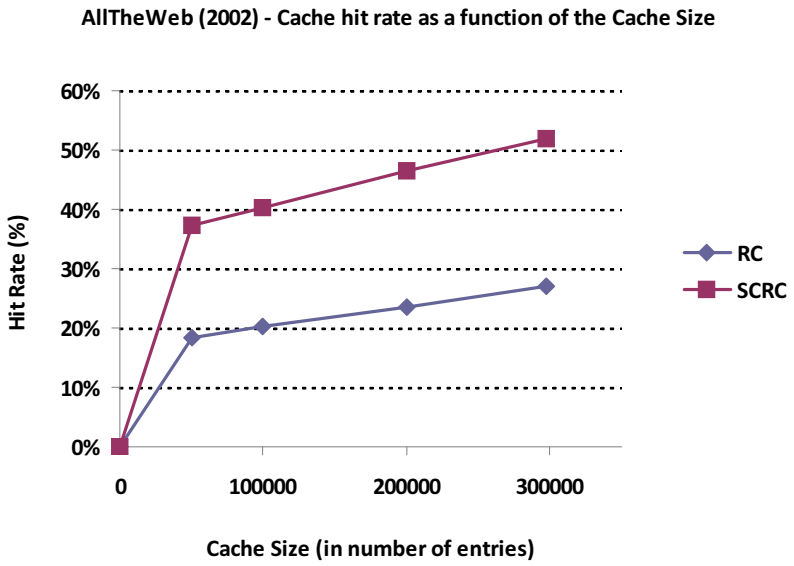


Figure 4.8: Hit Rate of the RC and the SCRC as a function of the Cache Size - AllTheWeb (2002) log

Table 4.10 shows the fraction of the identical hits, the exact set cover hits and the partial exact set covers that were obtained for each cache size over the AllTheWeb (2002) log.

Finally, we report our results over the log of the Altavista WSE. Figure 4.9 shows the cache hit rate of the RC and the SCRC over the Altavista query log. We see that the SCRC attains 14.4% - 20.5% higher hit rates than the RC.

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
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 4.10: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the AllTheWeb (2002) query log

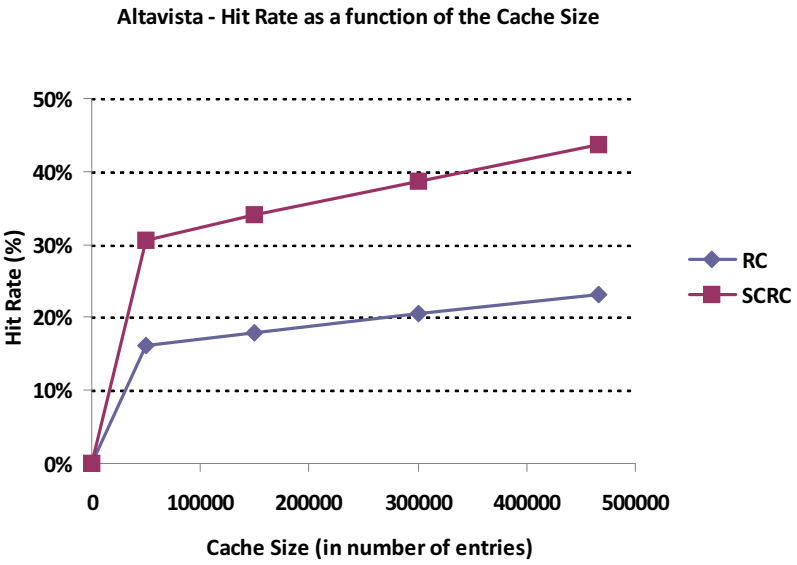


Figure 4.9: Hit Rate of the RC and the SCRC as a function of the cache size - Altavista log

Table 4.11 shows the fraction of the identical hits, the exact set cover hits and the partial exact set covers that were obtained for each cache size over the Altavista log. 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.

Cache Size	$IQR(Q_e, \cdot)$	$SCD(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$
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 4.11: Fraction of identical hits, exact set cover hits and partial exact set cover hits over the Altavista query log

4.1.5 Summary and Discussion

We studied and analyzed logs from 3 real web search engines. 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. By exploring the characteristics of these queries, we identified that most of these queries are on their majority short queries with few terms (most of them are composed of 2 terms). We also measured the size of the exact set covers for these queries and showed that most queries can be formulated through the disjoint union of 2 other queries. Using trace driven simulations for various cache sizes we showed that the SCRC outperformed the RC in all cases even when the cache size was set to a few cached entries. More precisely, we showed a SCRC can deliver up to 30% 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, there is a significant fraction of queries that constitute partial exact set covers and can potentially have a great impact on the performance of WSE by reducing the costs of the query evaluation process.

4.2 Experimental Evaluation over the Mitos WSE

In this section we describe the experiments that we have conducted over the Mitos WSE [38] and report our results. The scope of this evaluation was to identify the benefits of the SCRC in a real WSE. In Section 4.2.4, we consider the case where result caches store the complete answers of the cached queries and we propose and evaluate several cache filling strategies. Then, in Section 4.2.5.2, we consider the case where result caches store just the top-K results of the queries.

4.2.1 Hardware and Software Environment

The first version of Mitos was developed as a student project in the IR course (CS463) by undergraduate and graduate students of the Computer Science Department of the University of Crete in three semesters (spring: 2006, 2007 and 2008).

Mitos 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. The index based representation uses the HStore Object Relational representation described in [37]. This representation uses the PostgresSQL (PSQL) hstore data type for storing terms and their occurrences. More precisely, hstore is a data type for storing sets of (key,value) text pairs in a single PSQL data field. In our case, the key is the document id d and the value is the term frequency ($tf_{d,t}$).

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.

4.2.2 Document Collection

Our document collection is a set of web pages crawled by the Mitos web crawler. The index contains 240,108 distinct terms and the total postings of these terms are 5,039,486. The total size of the pages is 5.6 GB, yielding a DBMS-based index of approximately 170 MB and an inverted file of 52 MB without positional information.

4.2.3 Query Traces

Next, we describe both synthetic and real query traces that we used throughout the experimental evaluation over the Mitos WSE.

4.2.3.1 Synthetic Query Traces

Seed Query Set. We used a *seed query set* for the generation of the synthetic query sets comprising: 13 single word queries, 28 queries comprising two words, 1 query comprising three words, 5 queries with 4 words, 2 queries with 5 words, 1 query with 8 words. All the queries were normalized for case and white space and stopwords were evicted. The evaluation queries yielded 110 total query terms, comprising 81 distinct terms. The average query length of the evaluation queries is 2.2 terms. The maximum result set size of the evaluation queries is 10,934 documents, the minimum result set size is 20 documents and the average result set size is 4,063 documents.

In our experiments we used the following synthetic query traces:
(a) **Query Set A**: a stream of 10^4 queries randomly selected from the seed query set. This query stream has neither exact set covers nor partial exact set covers at all.
(b) **Query Set B**, a stream of 10^4 random queries generated as follows: From the 50 evaluation queries of the seed query set we chose the queries containing 2 terms. Then, for each query composed of 2 query terms, we generated 14 additional 3-term queries by merging each 2-term query with a random query term, selected from the single term queries and 14 additional 4-term queries by merging each 2-term query with another random 2-term query. Thus, we ended up with a total of 78 distinct queries. Finally, from these 78 queries, we produced a stream of 10^4 randomly selected queries.
(c) **Query Set C**: a stream of 10^4 random queries generated as follows: From the 50 evaluation queries of the seed query set we chose the queries containing 2 terms. Then, we generated 5 additional 3-term queries by merging each 2-term query with a random single term query, yielding totally 55 distinct queries. From this set of queries, we produced a stream of 10^4 randomly selected queries.
(d) **Query Set D**: a stream of 10^4 queries, randomly generated by combining index terms with characteristics more close to real query streams.

In the sequel, we will use the abbreviations Q_A, Q_B, Q_C and Q_D to denote the set of queries contained in Query Sets A, B, C and D respectively. Table 4.12 shows the values of the metrics for these sets.

Query Set	**Metrics**				
	$\|Q_e\|$	$IQR(Q_e)$	$SCD(Q_e)$	$PESCD(Q_e)$	$AVGQLEN(Q_e)$
Q_A	10,000	99.5%	0%	20%	2.2
Q_B	10,000	99.2%	35.9%	8.2%	2.66
Q_C	10,000	99.4%	10%	13.8%	1.98
Q_D	10,000	35%	35%	45%	2.4

Table 4.12: Query Set Metrics of Synthetic Traces

4.2.3.2 Real Query Traces

Moreover, we used two real query sets. More precisely, we used a trace of queries issued to the *Altavista* WSE on Sept. 9, 2002 and another trace of queries submitted to the *Excite* WSE on 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 [6].

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

Table 4.13 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 4.3).

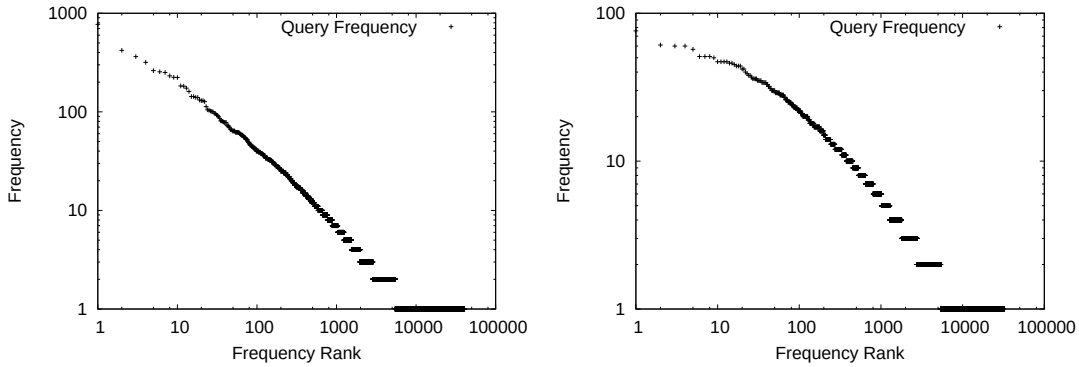
Query Set	**Metrics**				
	$\|Q_e\|$	$IQR(Q_e)$	$SCD(Q_e)$	$PESCD(Q_e)$	$AVGQLEN(Q_e)$
AltaVista	70,710	43.6%	35.3%	20.1%	2.0
Excite	50,000	36.4%	38.2%	38%	2.2

Table 4.13: Query Set Metrics of Real Traces

Figures 4.10 (a) and (b) show the distributions of the queries of the *Altavista* query set and the *Excite* query set respectively. The 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

[6] The same approach for filtering the queries has been also employed in [32].

normalized frequency for a given query. Both axes are on a log scale. The distribution of the query frequencies of these sets follow power-law distributions and we observe that there is a very long tail of queries that occur only once.



(a) *Altavista* query set (b) *Excite* query set

Figure 4.10: The distribution of the queries

4.2.4 SCRC

In this section, we evaluate the performance of the SCRC when the cache stores the complete answer of each cached query (all the results). We report results using the Okapi BM25 weighting scheme. Results for other models which are based on *decomposable* scoring functions are expected to be almost the same since the cache structure and the query evaluation algorithm do not change at all and the index is consulted only upon a cache miss.

4.2.4.1 Result Caches Filling

Regarding result caches (RC and SCRC), we fill them randomly. Experiments using other cache filling policies are given in Section 4.2.4.4. Each element of $Ans_{cache}(q_c)$ in the plain RC consists of pairs of the form $(d_i, Score(d_i, q_c))$.

4.2.4.2 Posting List Cache Filling

Each uncompressed posting list $I(t)$ of a term t in the posting lists cache consists of pairs of the form $(d_i, tf_{d_i,t})$. We select the terms to be cached, using the Q_{TFDF} scheme. This is

motivated by the experiments that we have conducted. Specifically we experimented with three different static (offline) caching strategies:

- Q_{TF}: the scheme proposed by Baeza-Yates and Saint-Jean [10] (referred as Q_{TF} in [5]), which suggests caching the terms with the highest query-term frequencies.
- Q_{TFDF}: the scheme proposed by Baeza-Yates et al 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 and $df(t)$ is the document frequency of term t.
- MIN_{DF}: which selects the terms with the lowest document frequency. This means that terms are selected for caching in ascending order of their document frequency.

Figures 4.11 (a-b) show the cache miss ratio of a static posting lists cache for various cache sizes over query sets Q_B and Q_C respectively.

The Q_{TF} algorithm captures the repetition of the query terms but it does not consider the document frequencies of the query terms (popular terms have also long posting lists). Thus, few terms with their corresponding posting lists fit in the available cache space, thus the cache miss ratio is still high. In our case, the MIN_{DF} outperforms the Q_{TF} algorithm, since more terms fit the available cache space and terms do not present high repetition in the evaluation sequence. The performance of the Q_{TFDF} algorithm is very similar to the performance of the MIN_{DF} algorithm.

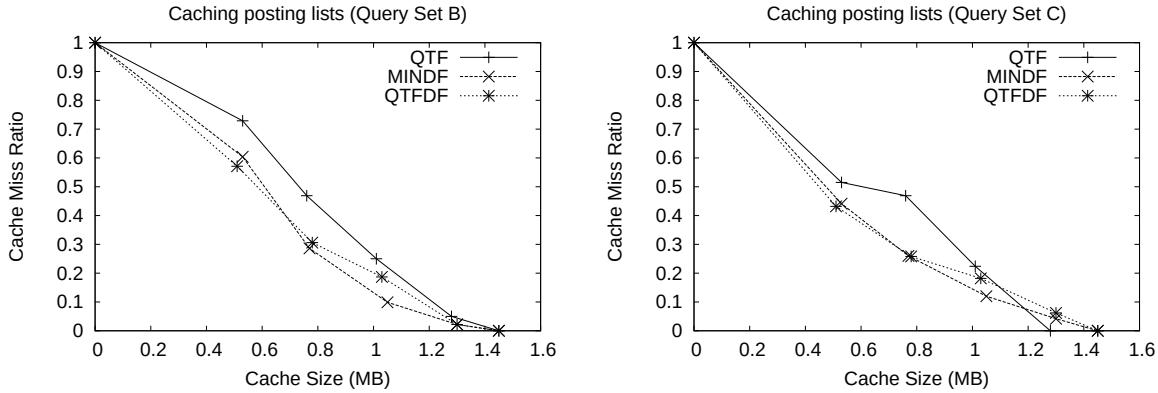


Figure 4.11: Cache miss ratio of different posting lists caching policies

4.2.4.3 Experimental Analysis

We measured the average query response time over Q_B and Q_C when the cache memory size is equal to 2.5% of the index size for the following cases:

(a) *N*o *C*ache (referred in figures as NC_B, NC_C),

(b) *P*osting *L*ist *C*ache (PLC_B, PLC_C) filled according to Q_{TFDF} scheme,

(c) *R*esult *C*ache (RC_B, RC_C), and

(d) SCRC ($SCRC_B$, $SCRC_c$).

We experimented with 2 ranking schemes:

- the Okapi BM25 model
- a global ranking model which assigns only static scores to the documents (i.e Pagerank scores). The score of each document d is simply its query-independent score ($Score(d, q) = g(d)$).

We made experiments for both index representations:

Figure 4.12(a-b) shows the results for the DBMS index and the typical inverted file (IF) index over the Q_B query set.

Figure 4.12(c-d) shows the corresponding results over the query stream Q_C.

The results show that the use of any caching mechanism always reduces the average query response time of the Mitos WSE. SCRC offers a significant speed up when documents are ranked according to the Okapi BM25 model. Tables 4.14 and 4.15 report the speedup for each case. The *speedup* S is defined by the following formula:

$$S = \frac{T_1 - T_2}{T_1} * 100$$

where T_1 is the execution time to evaluate a query when no caching mechanism is applied and T_2 is the execution time when applying a caching mechanism.

Retrieval Model	**Speedup on Average Query Evaluation Time (Q_B)**					
	DBMS			**IF**		
	PLC	RC	SCRC	PLC	RC	SCRC
Global Ranking	32.8%	**59.3%**	55.6%	49%	**55.5%**	35 %
Okapi BM25	15.2%	43.7%	**96.4%**	24%	40.5%	**72%**

Table 4.14: Speed up of PLC, RC and SCRC over Q_B

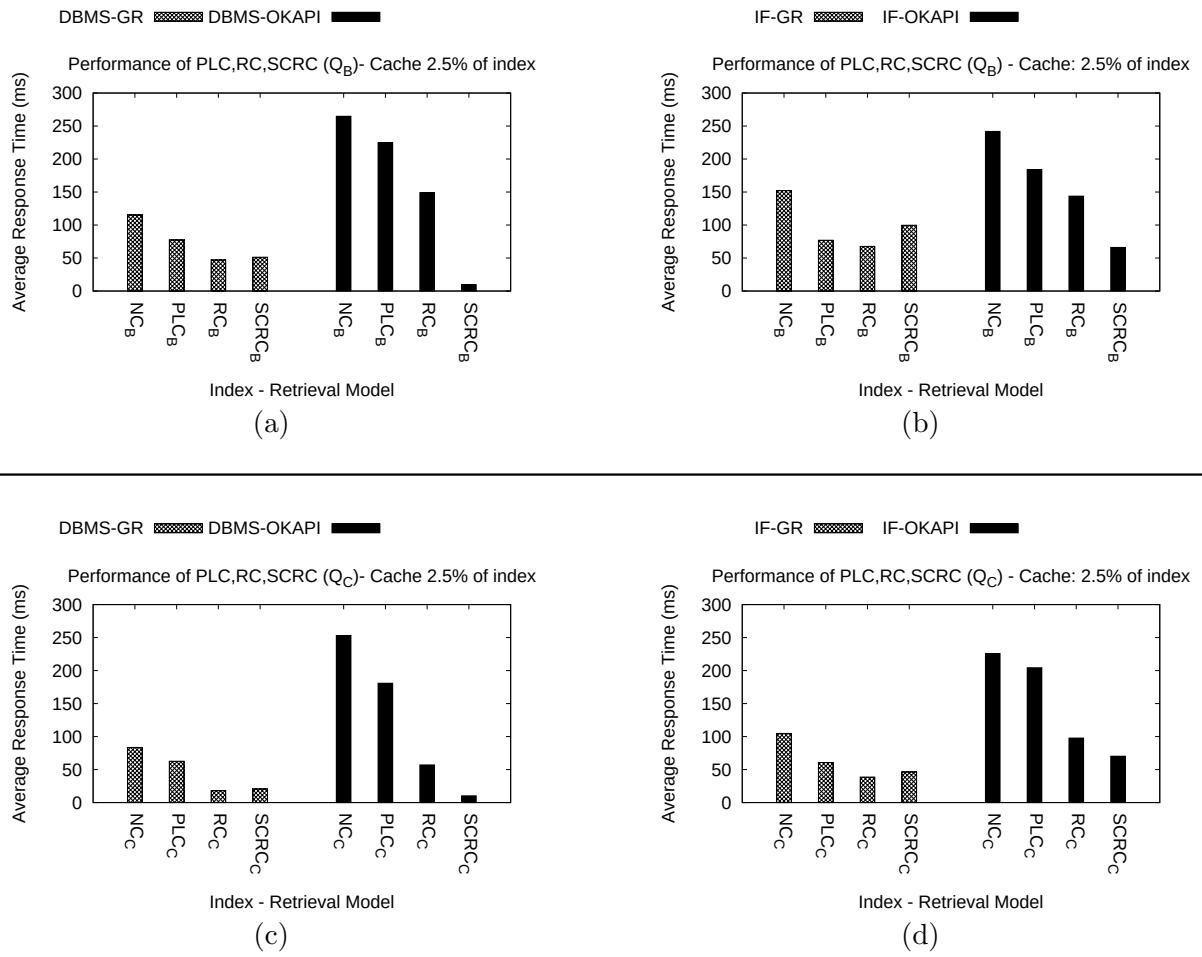


Figure 4.12: Average response with cache size = 2.5% of the index size (for inverted files and DBMS)

Retrieval Model	**Speedup on Average Query Evaluation Time (Q_C)**					
	DBMS			**IF**		
	PLC	RC	SCRC	PLC	RC	SCRC
Global Ranking	25%	**78.3%**	75.3%	40%	**62%**	60%
Okapi BM25	28.6%	77.4%	**95.9%**	25.1%	56.7%	**68.9%**

Table 4.15: Speed up of PLC, RC and SCRC over Q_C

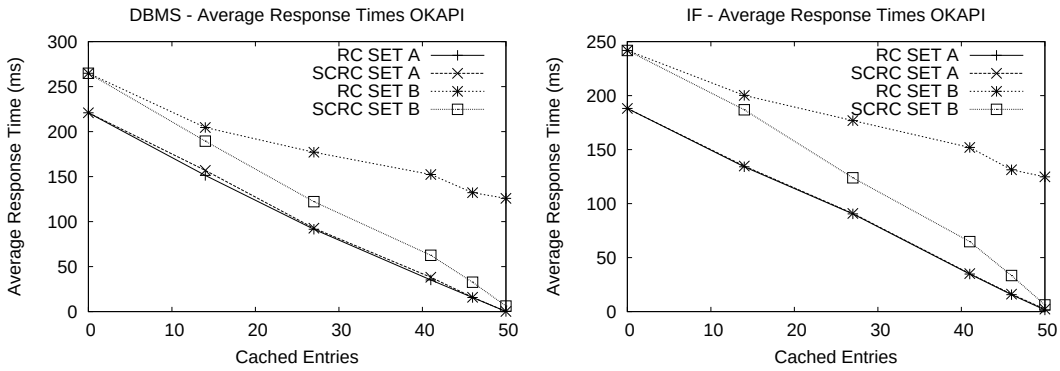
We observe that SCRC achieves the highest speedup for the Okapi BM25 model, when using the DBMS-index over both query sets (Q_B, Q_C). For instance, SCRC achieves speedup ranging from 72% to 96%, while RC offers at most 80% and PLC at most 29% speedup. Regarding, query-independent ranking (rows *Global Ranking*), RC slightly prevails, because no scores have to be computed and the benefits of SCRC are compensated by its higher cache miss time. Finally, it's worth noticing the PLC is the slowest in all cases. We have to stress that RC and SCRC outperformed PLC, even though in RC and SCRC the cached queries were selected randomly, while in PLC using the Q_{TFDF} policy. Obviously, a non random selection would further increase the relative benefits of RC and SCRC. The IQR of the query set was high and this at first glance seems to favor RC (and thus SCRC). However PLC also exploits the fact that the seed query set is small since it is based on Q_{TFDF} (which exploits popularity) and the number of the distinct terms in the query stream is small. This is evident also in Figure 4.11 where we can see that the miss ratio of PLC is very small (specifically 0.022 over Q_B and 0.062 over Q_C for size 2.5% of the index size). Finally, we have to mention that the benefits of the partial set cover hits were not very evident in our experiments because queries contained very few words. However, the speedup is expected to be higher for queries with several words.

4.2.4.4 Experiments for Various Cache Sizes

Now we report comparative results for SCRC and RC not with respect to cache size but we respect to the *number of the cached entries*. We experimented with two query sets: Q_A and Q_B, for Okapi BM25 retrieval model. Figures 4.13(c-d) show the average response time for both query sets when the index is Inverted File based and DBMS-based respectively. Notice that in both figures the plots of RC and SCRC that correspond to query set Q_A almost coincide. RC is slightly faster as expected since the miss time in the SCRC is higher than in the RC cache and $SCD(Q_A) = 0$. Over Q_B, SCRC is faster in all cases.

4.2.4.5 Cache Filling Strategies

Here we describe methods that take as input a query log and fill a SCRC. These methods are appropriate for caches hosting all the results of a query. Our objective is to maximize the average query response speedup. Given a query log Q_e, we can identify the following



(c) (d)

Figure 4.13: Average response time wrt number of cache entries

strategies for populating Q_c:

- **RC-like**

  Select the queries with the highest frequency. Order the queries of $Set(Q_e)$ with respect to their frequency and fill the cache starting from the queries with the highest frequency. The frequency of a query q in a stream Q_e is defined as

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

  This policy aims at maximizing the gain from identical hits.

- **PLC-like**

  Given two queries $q, q' \in Set(Q_e)$ we shall write $q \sqsubseteq q'$ if $t(q) \subset t(q')$. One policy is to order the queries with respect to $\sqsubseteq$, i.e. define a partial order $(Set(Q_e), \sqsubseteq)$. Then start filling the cache by traversing the Hasse Diagram of $(Set(Q_e), \sqsubseteq)$ starting from the minimal elements and going upwards level-wise. It is like performing a Breadth-First-Search starting from the minimal elements (an example is given later on). It is not hard to see that this policy aims at maximizing the gain from set cover hits. For instance if all minimal elements of $(Set(Q_e), \sqsubseteq)$ are placed in the cache, then every query in Q_e can be answered either by an identical hit or a set cover hit. If a query stream comprises a lot of single word queries, and only these are placed in the cache, then the resulting cache would be like a PLC.

- **FreqByAnswerSize**

  Order the queries of $Set(Q_e)$ with respect to their profit and fill the cache starting from the queries with the highest profit. The profit of a query q in a stream Q_e is defined as

$$FreqByAnswerSize(q, Q_e) = \frac{freq(q, Q_e)}{|Ans(q)|}$$

  This policy aims at utilizing the cache space effectively and maximizing the gain from identical hits.

- **SiPoCo**

  We need a more sophisticated policy which should increase the probability of identical hits *and* of set cover hits. At the same time it should demote queries whose answers are big and thus waste a lot of cache space. Now to promote queries that can contribute to set covers, we define a measure, called *SiPoCo* (from Size, Popularity, Coverage), as:

$$SiPoCo(q) = \sum\{\ FreqByAnswerSize(q')\ |\ q \sqsubseteq q'\}$$

  So we count the value also of the queries to which q could contribute to.

As an example, consider a stream Q_e where the elements of $Set(Q_e)$ are those listed in the first column of Table 4.16.

q	$freq(q)$	$\|Ans(q)\|$	$FreqByAnswerSize(q)$	$SiPoCo(q)$
q_1: {a}	2	10	0.2	0.2 + 0.2 +0.1 + 0.4 + 0.25 = 1.15
q_2: {d}	2	20	0.1	0.1 + 0.25 = 0.35
q_3: {a, b}	2	10	0.2	0.2 + 0.4 = 0.6
q_4: {a, c}	2	20	0.1	0.1 + 0.4 + 0.25 = 0.75
q_5: {a, b, c}	4	10	0.4	0.4
q_6: {a, c, d}	5	20	0.25	0.25

Table 4.16: Cache Filling Strategies - An example

Below we see the ordering derived by each criterion (Freq, PLC-like, FreqByAnswerSize, SiPoCo):

$$
\begin{aligned}
Freq &: \langle q_6, q_5, \{q_4, q_3, q_1, q_2\}\rangle \\
&= \langle \{a,c,d\}, \{a,b,c\}, \{\{a,c\}, \{a,b\}, \{a\}, \{d\}\}\rangle \\
PLC-like &: \langle \{q_1, q_2\}, \{q_3, q_4, q_6\}, q_5\} \\
&= \langle \{\{a\}, \{d\}\}, \{\{a,b\}, \{a,c\}, \{a,c,d\}\}, \{a,b,c\}\}\rangle \\
FreqByAnswerSize &: \langle q_5, q_6, \{q_3, q_1\}, \{q_4, q_2\}\rangle \\
&= \langle \{a,b,c\}, \{a,c,d\}, \{\{a,b\}, \{a\}\}, \{\{a,c\}, \{d\}\}\rangle \\
SiPoCo &: \langle q_1, q_4, q_3, q_5, q_2, q_6\rangle \\
&= \langle \{a\}, \{a,c\}, \{a,b\}, \{a,b,c\}, \{d\}, \{a,c,d\}\rangle
\end{aligned}
$$

Firstly, we observe that each cache filling strategy produces a different ordering of the queries. At a first glance, the *PLC-like* strategy tends to promote short term queries. Strictly speaking, the *SiPoCo* strategy does not guarantee that queries contributing to set covers will be ranked in the first positions. For instance, in the above query stream queries q_2 and q_4 cover query q_6, but they are not ranked both in the first positions of the ordered list that *SiPoCo* produces. More, notice that a query with the highest $freq(q)/|Ans(q)|$ ratio will be always ranked first by the *FreqByAnswerSize* strategy (in our example query q_5), but this is not the case for the *SiPoCo*. In the above example, q_6 is ranked in the last position of the list produced by the $SIPOCO$ strategy. Query q_5 does not contribute to set covers and is ranked lower in the final ordering than other queries.

Here we report some experiments of non-random cache filling policies using the Okapi BM25 retrieval model. We filled the caches using the above filling policies. Regarding the RC, we fill it using the most frequent queries.

Figure 4.14 shows the average response times of the Mitos WSE when employing these techniques.

We observe that the most beneficial cache filling strategy for the SCRC is the $SIPOCO$ strategy, because it captures both *identical* and *set cover* hits. When employing this policy, SCRC is up to 3 times faster than RC (for large cache sizes) and up to 4 times faster than PLC on the Okapi BM25 model.

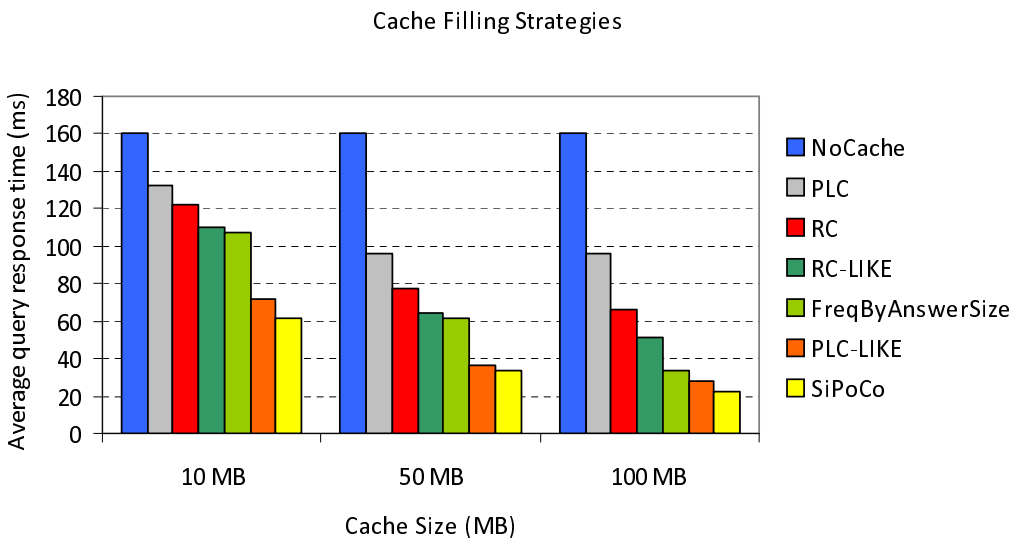


Figure 4.14: Cache Filling Strategies - Average Response Times

Table 4.17 reports in detail the fraction of the identical hits, the partial exact set cover hits ($PESCD$) and the exact set cover hits (SCD) that each cache filling strategy achieves for each cache size. (The statistics of the plain results cache are depicted in the last row of each sub-table.)

From Table 4.17, we observe that the $FreqByAnswerSize$ strategy achieves the highest number of identical hits since it allows the cache to host the maximum number of queries with their answers. Its drawback is that it ignores queries that are likely to contribute to set covers. More, we observe that even though the number of repeated queries is very high, the $RC-LIKE$ strategy is always less effective than the $PLC-LIKE$ and the $SiPoCo$ strategy. This is expected, since the $RC-LIKE$ strategy does not exploit set cover queries, and does not consider the size of the query answers as a criterion. Thus, it does not exploit suitably the available cache space. We also observe that the $PLC-LIKE$ strategy always achieves the max number of partial set covers. However, it does not consider at all the frequency of the queries and for this reason its efficiency in answering identical queries is very limited. Finally, we see that the $SiPoCo$ strategy leverages the advantages of the former strategies, since it captures a large fraction of identical queries and can answer also a sufficient fraction of queries derived through set cover hits.

Metrics			
$M = 10$ **MB**			
Policy	$IQR(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$	$SCD(Q_e, \cdot)$
Random	5.6%	6.79%	0.11%
RC-like	11.05%	11.45%	1.45%
PLC-like	9.74%	52.6%	18.69%
FreqByAnswerSize	19.13%	41.95%	3.89%
SiPoCo	12.1%	19.25%	30.61%
RC	11.05%	-	-
$M = 50$ **MB**			
Policy	$IQR(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$	$SCD(Q_e, \cdot)$
Random	27.61%	21.22%	1.49%
RC-like	39.56%	22.50%	2.0%
PLC-like	27.78%	42.81%	29.66%
FreqByAnswerSize	52.3%	29.53%	8.71%
SiPoCo	50.18%	26%	21.07%
RC	39.56%	-	-
$M = 100$ **MB**			
Policy	$IQR(Q_e, \cdot)$	$PESCD(Q_e, \cdot)$	$SCD(Q_e, \cdot)$
Random	47.31%	22.62%	3.65%
RC-like	47.31%	22.62%	3.65%
PLC-like	48.01%	32.58%	19.66%
FreqByAnswerSize	75.38%	15.12%	6.61%
SiPoCo	74.45%	12.88%	11.78%
RC	47.31%	-	-

Table 4.17: Cache Filling Strategies: Comparative Evaluation of cache metrics over Q_D

4.2.5 Top-K SCRC

Motivated by the fact that most users view only the first pages of results when submitting a query in a WSE, we now describe the comparative evaluation of a Top-K SCRC with a Top-K RC, where only the top-K answers are cached. For this evaluation, we used the real query traces in order to validate our results. Preliminary experiments showed that the performance gain achieved either by a SCRC or a RC is roughly the same in both releases (dbms-based release and inverted file release), so we report results obtained from the dbms-based release of Mitos .

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 fraction of the queries that can be 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.

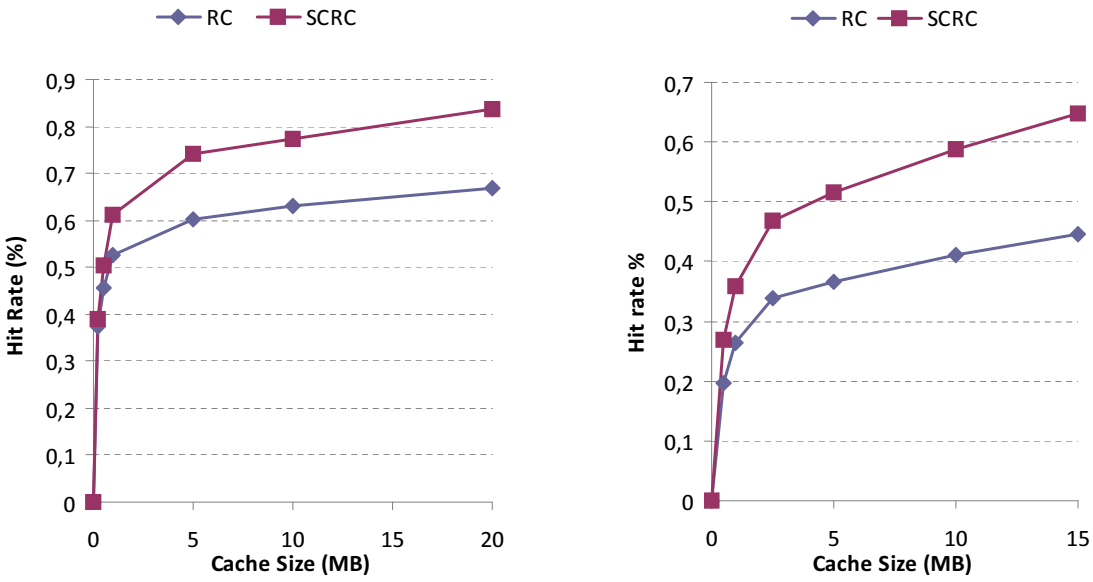
4.2.5.1 Result Caches Filling

Regarding the initialization of the caches, we follow the same strategy as the one described in Section 4.1.4.4. We split each query trace into 2 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-K answers ($K = 100$). 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 a cached answer (denoted by $Ans_{cache}(q_c)$) consists of pairs of the form $(d_i, Score(d_i, q_c))$; hence, both caches retain the same content.

4.2.5.2 Experimental Analysis

The first kind of experiments aims to assess the hit rate of each caching mechanism and the second kind of experiments aims to measure the average query execution time when the WSE employs either the SCRC or the RC.

Recall that in a RC, a query is considered to be a *cache hit*, iff there is an *identical* query in the cache. In a SCRC, a query is considered to be a *cache hit*, iff there is either an *identical* query in the cache or a *set cover* of the query in the cache. Figures 4.15 (a) and (b) illustrate the cache hit rate of the RC and the SCRC over the *Altavista* and the *Excite* query set respectively.



(a) *Altavista* query set

(b) *Excite* query set

Figure 4.15: Top-K SCRC Vs Top-K RC: Hit Rate as a function of the Cache Size

We observe that as the cache capacity increases, the hit rate of the RC and the SCRC also increase. Over all query sets, the SCRC achieves higher hit rates, since always a significant fraction of the submitted queries are resolved by the SCRC as *set cover hits*. More precisely, over the *Altavista* query set, the SCRC answers 16.8% more queries than the RC. Over the *Excite* query set, the SCRC achieves up to 20% higher hit rates than the RC. Note that in the SCRC case, although partial exact set covers ($PESC$) are considered as misses (since

we must access also the main index), they are likely to contribute to the reduction of the average response time per query as we verify in the next section.

Next we report the average query response times of the Mitos WSE when it employs either the RC or the SCRC for various cache sizes over the Altavista query set and the Excite query set. We conducted this second kind of experiments in order to verify that the higher hit rate of the SCRC also results to lower query response times of the WSE. Whenever a query cannot be answered by the cache, then the cache reports a miss and the query is fully evaluated from the main index.

Altavista query set: Figure 4.16 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 when using the RC and the SCRC respectively.

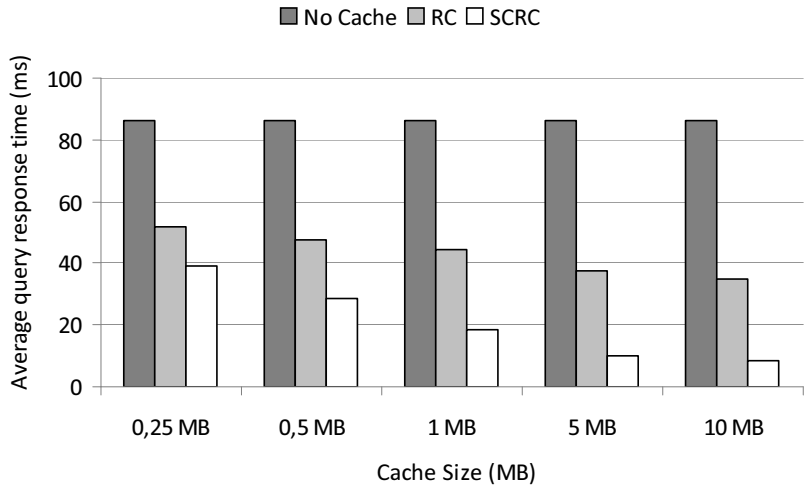


Figure 4.16: Average query response time over the *Altavista* query set (in ms)

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 $M = 5\ MB$), SCRC is at least 2 times faster than the RC. When the cache size is large (i.e $M = 10\ MB$, $M = 15\ MB$), SCRC is 4 times faster than the RC.

Table 4.18 shows in detail the fraction of the identical queries, the exact set covers and the partial exact set covers for each cache size. As in the analysis in Section 4.1.4.4, we observe that as the cache size increases: the fraction of the identical hits and the exact set cover hits increase and the fraction of the partial exact set covers decrease. Recall that a RC and a SCRC achieve the same fraction of identical hits.

Moreover, we observe that as the number of the exact set cover hits increase, 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.

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 4.18: Fraction of identical hits, exact set cover hits and partial exact set cover hits over query set *Altavista*

This is also evident in Table 4.19, where we report the average query execution time (in milliseconds) to answer a query of the *Altavista* query set through:

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

As expected, 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 also very fast. 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.

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 4.19: 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

Table 4.20 shows in detail the average execution time (in microseconds) spent for answering a query through a set cover hit. We report the execution time which is spent in order to:

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

We observe that most of the time is spent on aggregating the answers of the cached sub-queries that cover the incoming query. The average time for generating the sub-queries of each query (using the *hash-based* approach presented in Section 3.5.1) 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.

Excite query set: Figures 4.17 illustrates the average query response time of the Mitos WSE when employing the RC and the SCRC over the *Excite* query set.

We experimented also with very small cache sizes in order to verify that the benefits of

Cache Size	Lower Time	Greedy Time	Aggregation Time	Accuracy Time
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 4.20: Detailed execution times (in microseconds) for deriving the answer of a query through a set cover hit over the *Altavista* query set

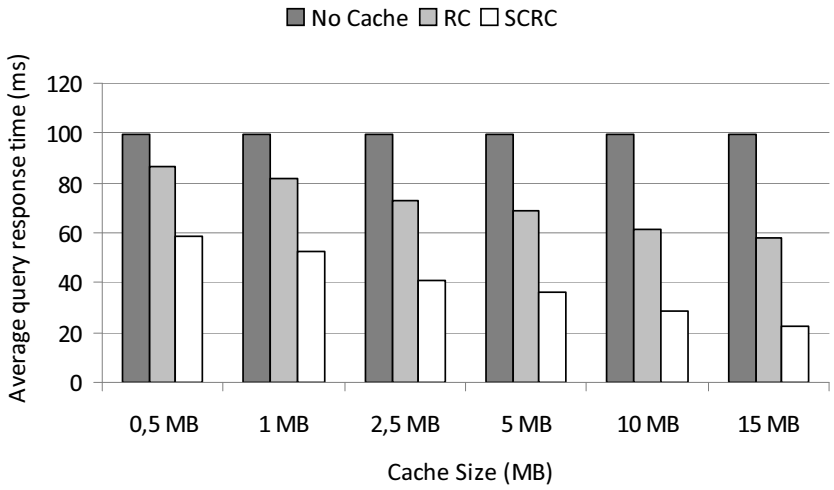


Figure 4.17: Average query response time over the *Excite* query set

the SCRC are also evident and are not compensated by its higher miss time, compared to the SCRC. 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.

Table 4.21 shows in detail the fraction of the identical queries, the exact set covers and the partial exact set covers for each cache size.

Table 4.22 reports in detail the average query execution time (in milliseconds) to answer a query of the *Excite* query set through:

- the index

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

- an identical cached query
- an exact set cover (ESC)
- a partial exact set cover (PESC)

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 requires 50% less time than evaluating it from the main index.

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 4.22: 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

Table 4.23 reports in detail the average time (in microseconds) spent to answer a query through a set cover hit. 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 Time	Greedy Time	Aggregation Time	Accuracy Time
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 4.23: Detailed execution times (in microseconds) for deriving the answer of a query through a set cover hit over the *Excite* query set

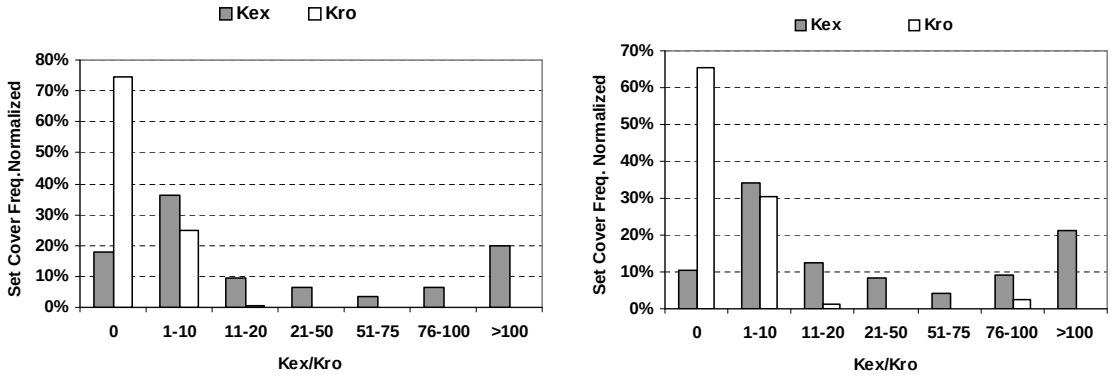
4.2.5.3 Estimating the accuracy of the composed top-K answer

We also measured the accuracy of the composed top-K answers for those queries that were answered through *set cover hits*, when M is set to 10 MB; results for other values of M are very similar. For measuring the accuracy of these answers we used two different approaches, as we describe next.

Mathematically guaranteed accuracy: Firstly, we estimated the minimum accuracy of the composed top-K answer using the approach presented in Section 3.6. Recall that this approach guarantees that the first K_{ex} elements are *definitely* the highest scored elements, and that the first K_{ro} elements have the right relative order. For the rest of the elements in the composed answer this approach cannot guarantee, despite the fact that they are likely to be included among the top scored elements.

The distribution of the K_{ex} and K_{ro} values over the *Altavista* query set and the *Excite* query set is illustrated in Figures 4.18 (a) and (b) respectively. The x-axis ranges $[0..B]$, where $|B|$ is the result set size. The y-axis illustrates the proportion of the queries that are answered through set cover hits and have the x value. Over these query sets, the average K_{ex} value is 43 and the average K_{ro} value is 3.

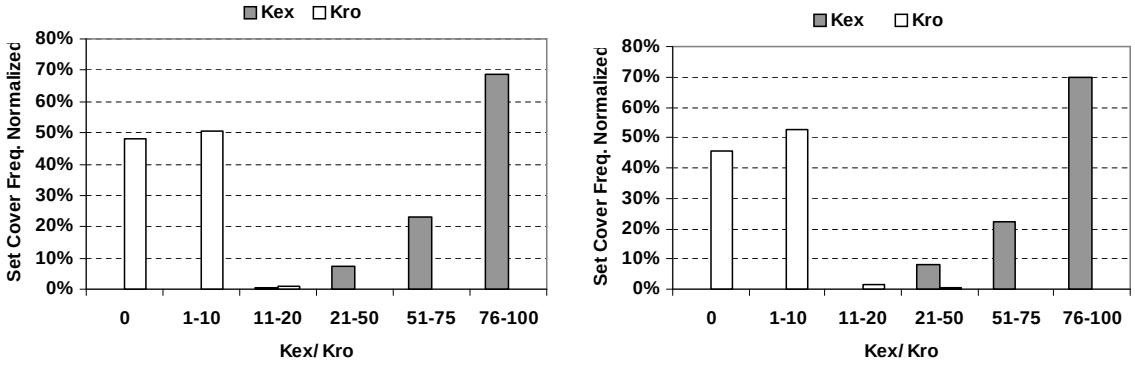
Using the answer of the main index: Next, using the same configurations (same M and K values) as before, we measured the actual accuracy of the composed top-K answer using the main index. For each query, which was answered through a *set cover hit* from the SCRC, we also evaluated it from the main index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



(a) *Altavista* query set (b) *Excite* query set

Figure 4.18: Mathematically guaranteed accuracy of the composed top-K answer for set cover hits

Figures 4.19 (a) and (b) show the accuracy of the composed top-K over the *Altavista* and the *Excite* query set respectively when using the answer of the main index.



(a) *Altavista* query set (b) *Excite* query set

Figure 4.19: Index-based accuracy of the composed top-K answer for set cover hits

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 twice as double in contrast to the previous approach (83 vs. 43). This means that the answer through a set cover hit contains 80% of the 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, when the answer is composed of 100 documents ($K = 100$). In addition, 38%

of these answers contain all the top matching documents (100 docs). These results are very encouraging since previous work [26, 43] on user's web search behavior has revealed 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.

Recall also that the accuracy of the query answers we are examining refer only to those queries that were answered from the cache as *set cover hits* (approximately 20% of all the queries submitted to the WSE) and not for the whole query stream.

Chapter 5

Concluding Remarks

We introduced SCRC, a novel scheme for results caching which bridges the gap between results caching and posting lists caching and aims at maximizing the utilization of a results cache. We proposed a query evaluation scheme where how upon a cache miss, the incoming query can still be very quickly evaluated if it is a disjoint union of the cached queries. To this end, we reduced the problem of finding the "best" cached sub-queries that are required for answering the incoming query, to the well-known Set Cover Problem.

We introduced the notion of additively decomposable scoring functions which determines the applicability of SCRC and we showed that several best-match retrieval models (e.g VSM, Okapi BM25 and several hybrid retrieval models) which are traditionally used in Information Retrieval and WSEs rely on such scoring functions.

Our study over real query logs of three different WSEs showed that there is a significant fraction of queries in real query streams that can be formulated as the disjoint union of the rest queries. Our analysis and experimental results over these logs revealed that a SCRC can deliver up to 30% higher hit rates than a plain RC.

We introduced and evaluated two flavors of the SCRC: one that stores the complete answers of the queries and another that stores the top-K answers. Regarding the former, we proposed and evaluated several static cache filling policies that maximize the speed up obtained by a SCRC and showed that the SCRC is 2 times faster than a plain RC and 3 times than a posting lists cache. Motivated by the fact that most users examine only the first pages of results, we introduced a variation of the SCRC, the Top-K SCRC and we

defined metrics for characterizing the quality of the answers derived through set cover hits. Our main results over real world query traces demonstrate the superiority of the SCRC over the RC in best-matching retrieval models (i.e Okapi BM25, VSM). More concretely, we showed that a Top-K SCRC is 2 times faster than a Top-K RC, while preserving the quality of the answers. We also showed that as the cache size increases the relative speedup of the RC against the SCRC also increases.

In bigger indexes the benefits of the SCRC are expected to be even higher, considering that the cost of query processing is dominated by the cost of traversing the inverted lists, which grow linearly with the collection size.

Bibliography

[1] Ismail Sengor Altingovde, Rifat Ozcan, and Özgür Ulusoy. A cost-aware strategy for query result caching in web search engines. In *ECIR '09: Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval*, pages 628–636, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] V.N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 372–379. ACM New York, NY, USA, 2006.

[3] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, New York, NY, USA, 2001. ACM.

[4] Ricardo Baeza-Yates. Web mining in search engines. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 3–4, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[5] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4):1–28, 2008.

[6] Ricardo A. Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[7] Ricardo A. Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, and Hans Friedrich Witschel. Admission policies for caches of search engine results. In *SPIRE*, pages 74–85, 2007.

[8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] Ricardo A. Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[10] Ricardo A. Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[11] Steven M. Beitzel, Eric C. Jensen, Abdur Chowdhury, David Grossman, and Ophir Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 321–328, New York, NY, USA, 2004. ACM.

[12] Roi Blanco and Alvaro Barreiro. Boosting static pruning of inverted files. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 777–778, New York, NY, USA, 2007. ACM.

[13] Roi Blanco and Alvaro Barreiro. Static pruning of terms in inverted files. In *ECIR*, pages 64–75, 2007.

[14] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[15] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434, New York, NY, USA, 2003. ACM.

[16] Eric W. Brown, James P. Callan, W. Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. In *EDBT '94: Proceedings of the 4th international conference on extending database technology*, pages 365–378, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[17] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[18] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Web mining: Information and pattern discovery on the world wide web. In *ICTAI*, pages 558–567, 1997.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[20] Edleno S. de Moura, Célia F. dos Santos, Daniel R. Fernandes, Altigran S. Silva, Pavel Calado, and Mario A. Nascimento. Improving web search efficiency via a locality based static pruning method. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 235–244, New York, NY, USA, 2005. ACM.

[21] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. 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.

[22] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 431–440, New York, NY, USA, 2009. ACM.

[23] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, Inc., Orlando, FL, USA, 1978.

[24] Bernard J. Jansen and Amanda Spink. An analysis of web documents retrieved and viewed. In *International Conference on Internet Computing*, pages 65–69, 2003.

[25] Bernard J. Jansen and Amanda Spink. How are we searching the world wide web?: a comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, 2006.

[26] Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. 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.

[27] Björn T. Jónsson, Michael J. Franklin, and Divesh Srivastava. Interaction of query evaluation and buffer management for information retrieval. *SIGMOD Rec.*, 27(2):118–129, 1998.

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

[29] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on World Wide Web*, pages 19–28. ACM New York, NY, USA, 2003.

[30] Ronny Lempel and Shlomo Moran. Optimizing result prefetching in web search engines with segmented indices. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 370–381. VLDB Endowment, 2002.

[31] Ronny Lempel and Shlomo Moran. Competitive caching of query results in search engines. *Theor. Comput. Sci.*, 324(2-3):253–271, 2004.

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

[33] Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li. Active query caching for database web servers. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 92–104, London, UK, 2001. Springer-Verlag.

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

[35] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *SIGIR*, pages 191–198, 2007.

[36] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Static query result caching revisited. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 1169–1170, New York, NY, USA, 2008. ACM.

[37] Panagiotis Papadakos, Yannis Theoharis, Yannis Marketakis, Nikos Armenatzoglou, and Yannis Tzitzikas. Object-relational database representations for text indexing. *CoRR*, abs/0906.3112, 2009.

[38] Panagiotis Papadakos, Giorgos Vasiliadis, Yannis Theoharis, Nikos Armenatzoglou, Stella Kopidaki, Yannis Marketakis, Manos Daskalakis, Kostas Karamaroudis, Giorgos Linardakis, Giannis Makrydakis, Vangelis Papathanasiou, Lefteris Sardis, Petros Tsialiamanis, Georgia Troullinou, Kostas Vandikas, Dimitris Velegrakis, and Yannis Tzitzikas. The anatomy of mitos web search engine. *CoRR*, abs/0803.2220, 2008.

[39] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[40] Stephen E. Robertson and Karen Sparck Jones. Relevance weighting of search terms. pages 143–160, 1988.

[41] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3. In *TREC*, pages 0–, 1994.

[42] Patricia Correia Saraiva, Edleno Silva de Moura, Rodrigo C. Fonseca, Wagner Meira Jr., Berthier A. Ribeiro-Neto, and Nivio Ziviani. Rank-preserving two-level caching for scalable search engines. In *SIGIR*, pages 51–58, 2001.

[43] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[44] Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, and Ricardo A. Baeza-Yates. Resin: a combination of results caching and index pruning for high-performance web search engines. In *SIGIR*, pages 131–138, 2008.

[45] Aya Soffer, David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, and Yoëlle S. Maarek. Static index pruning for information retrieval systems. In *SIGIR*, pages 43–50, 2001.

[46] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.

[47] Yinglian Xie and David R. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM*, 2002.

[48] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[49] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2), 2006.

Appendix

A.1 Proofs

Prop A.1.1

The typical scoring function of the Vector Space Model is not *decomposable*.

Proof:

We will prove this proposition by a counterexample. Consider a document collection that consists of only one document d = "barack obama white house" and the query q = "barack obama". We will show that :

$$Sim_{cos}(d, \text{“}barack\ obama\text{”}) \neq Sim_{cos}(d, \text{“}barack\text{”}) + Sim_{cos}(d, \text{“}obama\text{”})$$

Let's first compute $Sim_{cos}(d, q)$. The document d and the query q are represented by the following weighted vectors $\overrightarrow{d} = \{1, 1, 1, 1\}$ $\overrightarrow{q} = \{1, 0, 1, 0\}$ considering that the weights in the vectors correspond to the lexicographical order of the query terms. Here we have, $W_d = \sqrt{\sum_{t \in t(d)} w_{d,t}^2} = \sqrt{4} = 2$ and $W_q = \sqrt{\sum_{t \in t(q)} w_{q,t}^2} = \sqrt{2}$.
The score of the document d w.r.t q is computed as

$$Sim_{cos}(d, q) = \frac{\sum_{t \in t(d) \cap t(q)} w_{d,t} \cdot w_{q,t}}{W_q \cdot W_d} = \frac{1 * 1 + 0 + 1 * 1 + 0}{\sqrt{2} * 2} = \frac{2}{2 * \sqrt{2}} = \frac{\sqrt{2}}{2}$$

Let's now compute the similarity scores of the document d w.r.t query terms t_1 = "*barack*" and t_2 = "*obama*" individually. The query vectors of terms t_1 and t_2 are $\overrightarrow{t_1} = \{1, 0, 0, 0\}$ and $\overrightarrow{t_2} = \{0, 0, 1, 0\}$ respectively. We have:

$$Sim_{cos}(d, t_1) = \frac{\sum_{t \in t(d) \cap t_1} w_{d,t} \cdot w_{q,t}}{W_q \cdot W_d} = \frac{1 * 1 + 0 + 0 + 0}{1 * 2} = \frac{1}{2}$$

and

$$Sim_{cos}(d, t_2) = \frac{\sum_{t \in t(d) \cap t_2} w_{d,t} \cdot w_{q,t}}{W_q \cdot W_d} = \frac{0 + 0 + 1 * 1 + 0}{1 * 2} = \frac{1}{2}$$

Hence,

$$Sim_{cos}(d, \text{“}barack\text{”}) + Sim_{cos}(d, \text{“}obama\text{”}) = \frac{1}{2} + \frac{1}{2} = 1 \neq \frac{\sqrt{2}}{2}$$

$\diamond$

A.2 Set Cover Problem

An instance (X,F) of the set covering problem, consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F: $X = \bigcup_{S \in F} S$. We say that a subset $S \in F$ covers its elements. The problem is to find a minimum-size subset $C \subseteq F$ whose members all of X [19]:

$$X = \bigcup_{S \in C} S$$

The greedy algorithm for the set covering problem from [19] is presented next.

Algorithm 4 GREEDY-SET-COVER (X, F)

1: $C \leftarrow \emptyset$
2: $U \leftarrow X$
3: **while** $U \neq \emptyset$ **do**
4: select an $S \in F$ that maximizes $|S \cap U|$
5: $U \leftarrow U - S$
6: $C \leftarrow C \cup \{S\}$
7: **end while**
8: **return** C

The greedy algorithm chooses sets according to one rule: at each stage it greedily chooses the set $S \in F$ which contains the largest number of uncovered elements. Repeat until all the elements of U are covered.

Since each iteration of the while loop in lines 2-6 adds a set $S \in F$ to the cover C, and S must cover at least one of the as yet uncovered elements of X, it is clear that the maximum number of iterations of the while loop is at most $\min(|X|, F|)$. The actual body of the loop can be implemented in time $\mathcal{O}(|X||F|)$ and so the overall complexity of Algorithm 4 is $\mathcal{O}(|X||F| \min(|X|, |F|))$.

An algorithm that returns near-optimal solutions is called an approximation algorithm. We say that an algorithm for a problem has an approximation ratio of $p(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $p(n)$ of the cost C^* of an optimal solution: $\max\left(\frac{C}{C^*} \leq p(n)\right)$. The cost of the set-covering is the size of C, which defines as the number of sets it contains.

According to [19], Algorithm 4 is a polynomial-time $p(n)$-approximation algorithm to the set covering problem, with a logarithmic approximation ratio, where $p(n) = H(\max\{|S| : S \in F\})$. That is as the size of the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution.

Since, $H(n) = \sum_{i=1}^{n} \frac{1}{k} \leq \ln(n) + 1$, we have that

$$p(n) \quad = \quad H(\max\{|S| : S \in F\}) \leq H(|X|) \leq \ln|X| + 1$$

Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results. Hence, the greedy algorithm returns a set cover that is not at most $\ln(n) + 1$ times larger than the optimal set cover. $\diamond$

Next, we provide an example which shows an instance of the set covering. We show that the greedy algorithm (Alg. 4) may not result in an optimal solution.

Example A.2.1 Figure 1 shows an instance of the set covering problem.

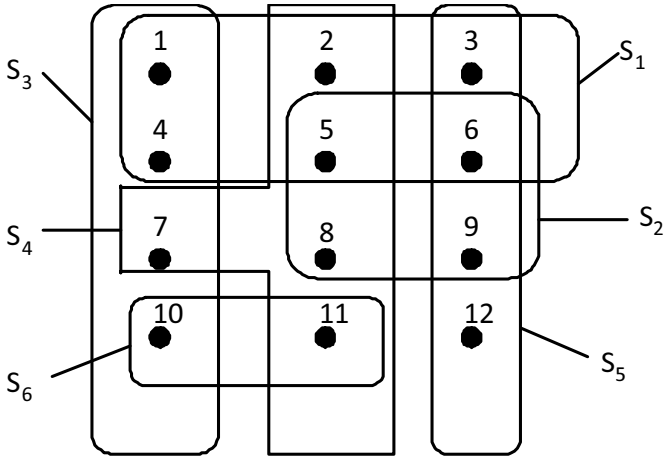


Figure 1: Set Cover Example

In this example $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and the family $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$ of subsets of X. The set cover in this example is $C = \{S_1, S_4, S_5\}$, since

$$\begin{aligned} \bigcup_{S_i \in C} S_i &= \{1, 2, 3, 4, 5, 6\} \cup \{2, 5, 7, 8, 11\} \cup \{3, 6, 9, 12\} \\ &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \\ &= X \diamond \end{aligned}$$

The set cover in this example is $C = \{S_3, S_4, S_5\}$. However, the greedy algorithm returns a different set cover, which is not the optimal (the one with the minimum size).

At the first iteration of the algorithm, it selects $S_1 = \{1, 2, 3, 4, 5, 6\}$, since it is the set that covers the most of the uncovered elements of X, which is six. Now $X = \{7, 8, 9, 10, 11, 12\}$ and $C' = \{S_1\}$ In the next iteration, the algorithm selects set $S_4 = \{2, 5, 7, 8, 11\}$ which covers 3 more elements (7, 8 and 11) and now $X = \{9, 10, 12\}$ and $C' = \{S_1, S_4\}$. Next, the algorithm selects set $S_5 = \{3, 6, 9, 12\}$, which covers 2 elements. Hence, now $X = \{10\}$ and $C' = \{S_1, S_4, S_5\}$. Finally, the algorithm selects set $S_3 = \{1, 4, 7, 10\}$ and terminates. The returned set cover is $C' = \{S_1, S_4, S_5, S_3\} \neq C$.

A.2.1 Greedy Exact Set Cover

In our context, there is not any guarantee that Algorithm 1 will surely return an exact set cover (ESC), even if one exists.

- The main reason that an ESC may not be returned is that the family F of the subsets of X (recall that in our case F corresponds to the cached sub-queries of q and X to the query terms of q) may not contain all the elements of X in order to form an exact set cover. Nevertheless, a partial exact set cover ($PESC$) will be definitely returned.

- Another reason is the use of the greedy algorithm (Alg. 1). The greedy algorithm may not return an ESC, even if one exists. However, we used also a non-greedy algorithm which examines all the combinations of the cached sub-queries of the incoming query and we reported in Section 4.1.4.1 that a non-greedy algorithm returns about 0.2% - 0.6% more exact set covers than Algorithm 1 over all the logs.

- Another reason that we may "miss" an exact set cover is due to the exact set cover restriction since, we are interested in set covers with pairwise disjoint subsets. For instance, if $X = \{$"a", "b", "c", "d"$\}$ and $F = \{S_1 =$ "$a\ b\ c$", $S_2 =$ "$a\ b$", $S_3 =$ "$c\ d$"$\}$, then Algorithm 1 will select S_1, since it covers most of the elements of X, but in the next step it will terminate (since either S_2 or S_3 are not pairwise disjoint with S_1) and will return a partial exact set cover ($PESC$). Instead, in the plain set cover problem where the sets of the cover may overlap (there is no restriction for exact set covers), Algorithm 4 will find a plain set cover, by selecting in order sets S_1 and S_3.

Next, we investigate the quality of the approximation returned by the greedy algorithm.

Let U be the query terms. 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 $|Rem(U)| = 0$. Recall that this decision problem in NP-Complete.

Let $Rem_F(U)$ be the uncovered elements of U if we run the greedy algorithm. We would like to cover all elements of U as this allows answering the incoming query from cached queries. Therefore

for evaluating the quality of approximation returned by the greedy algorithm, we could 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 then $Z > 0$. Obviously it also holds $Z \leq 1$, and 1 is the worst value for Z (no elements of U are covered). However we will refine the upper bound later on.

Let $F = \{S_1, \ldots, S_k\}$ be the 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.

- **Case $|U| = 3$**
  Here Z can be greater than zero. For example, let $F = \{\{a, b\}, \{a\}, \{b, c\}\}$. An ESC exists, but the greedy could fail (if it selects $\{a, b\}$ at its first iteration). Here we have $Z = 1 - 2/3 = 1/3$. Let $U = \{a, b, c\}$. An ESC can be one of the following:

  1. abc
  2. ab c
  3. a bc
  4. ac b
  5. a b c

  In the case 1 and 5 greedy will not fail. In cases 2, 3, 4 it can fail (2: if F contains ac, 3: if F contains ab, 4: if F contains ab). So 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. So the expected value of Z (if we assume that an ESC always exists) is 2/5*0 + 3/5 * 1 = 3/5 = 0.6. Summarizing, for $|U| = 3$ the probability that the greedy algorithm finds an ESC if it exists is 0.6 (assuming a uniform distribution).

Since from the query log analysis presented in previous chapter, we know the probabilities of $|U| = 1$, $|U| = 2$ and $|U| = 3$ in real query logs, we can compute the expected Z for those parts

of the query logs. Specifically, from the analysis of query logs we know that $P(|U| \leq 3) = 0.867$. For these queries the expected Z is $E(Z) = P(|U| = 1) * 0 + P(|U| = 2) * 0 + P(|U| = 3) * 0.6 = 0.277 * 0 + 0.38 * 0 + 0.21 * 0.6 = 0.126$.

Let's now discuss the general case, that includes cases where $|U| > 3$. Since an ESC exists there is certainly 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 $Z = (|U| - m - 0)/|U| = (|U| - m)/|U| = 1 - \frac{m}{|U|}$.

Although we have proved for all cases where $|U| \leq 3$ that if an ESC exists then the greedy algorithm will cover at least $\lceil \frac{|U|}{2} \rceil$ elements holds, it does not hold in the general case. In the general case, it will cover $|U| - m$ elements, where $m = max|S_i|$. Consider the following counterexample (Ex. A.2.2).

Example A.2.2 Assume $U = \{1, 2, 3, 4, 5, 6, 7\}$ and $F = \{S_1, S_2, S_3, S_4\}$, where

$$S_1 = \{1, 4, 7\}$$
$$S_2 = \{1, 2, 3\}$$
$$S_3 = \{4, 5, 6\}$$
$$S_4 = \{7\}$$

An exact set cover C in this example is $C = \{S_2, S_3, S_4\}$. However, the greedy algorithm will not return it. It will select set S_1 in its first iteration, since it covers most of the elements of U (3 elements) and then it will terminate, since all the remaining subsets are not pairwise disjoint with S_1 (it holds that $S_1 \cap S_j \neq \emptyset$, for each $j = 2, 3, 4$). In this case, the greedy algorithm will cover $m = 3$ elements, hence $Z = 1 - \frac{3}{7} = 0.57$.

A.2.2 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 A.2.3 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 3.2 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:

$$\begin{aligned} Sim'_{cos}(d, q_{c_1}) &= \frac{1*1+0*0+1*1+0*0}{\sqrt{2}} = \frac{2}{\sqrt{2}} = \sqrt{2} = 1.41 \\ Sim'_{cos}(d, q_{c_2}) &= \frac{1*0+1*1+1*1+1*1}{\sqrt{2}} = \frac{3}{\sqrt{2}} = 3 * \frac{\sqrt{2}}{2} = 2.12 \end{aligned}$$

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 $Ans(q)$ using $Ans_C(q)$. As we described in Section 3.4, the answer of q denoted by $Ans_C(q)$, is the union of the documents in the answers of the queries in C, Hence, $Ans_C(q) = d$. The score $Score_C(d, q)$ 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:

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

$\diamond$