IMPROVING THE EFFICIENCY OF SEARCH ENGINES: STRATEGIES FOR FOCUSED CRAWLING, SEARCHING, AND INDEX PRUNING

A DISSERTATION SUBMITTED TO

THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

İsmail Sengör Altıngövde

July, 2009

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Özgür Ulusoy (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Adnan Yazıcı

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Fazlı Can

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Enis Çetin

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

IMPROVING THE EFFICIENCY OF SEARCH ENGINES: STRATEGIES FOR FOCUSED CRAWLING, SEARCHING, AND INDEX PRUNING

İsmail Sengör Altıngövde
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Özgür Ulusoy
July, 2009

Search engines are the primary means of retrieval for text data that is abundantly available on the Web. A standard search engine should carry out three fundamental tasks, namely; crawling the Web, indexing the crawled content, and finally processing the queries using the index. Devising efficient methods for these tasks is an important research topic. In this thesis, we introduce efficient strategies related to all three tasks involved in a search engine. Most of the proposed strategies are essentially applicable when a grouping of documents in its broadest sense (i.e., in terms of automatically obtained classes/clusters, or manually edited categories) is readily available or can be constructed in a feasible manner. Additionally, we also introduce static index pruning strategies that are based on the query views.

For the crawling task, we propose a rule-based focused crawling strategy that exploits interclass rules among the document classes in a topic taxonomy. These rules capture the probability of having hyperlinks between two classes. The rule-based crawler can tunnel toward the on-topic pages by following a path of off-topic pages, and thus yields higher harvest rate for crawling on-topic pages.

In the context of indexing and query processing tasks, we concentrate on conducting efficient search, again, using document groups; i.e., clusters or categories. In typical cluster-based retrieval (CBR), first, clusters that are most similar to a given free-text query are determined, and then documents from these clusters are selected to form the final ranked output. For efficient CBR, we first identify and evaluate some alternative query processing strategies. Next, we introduce a new index organization, so-called cluster-skipping inverted index structure (CS-IIS). It is shown that typical-CBR with CS-IIS outperforms previous CBR strategies

(with an ordinary index) for a number of datasets and under varying search parameters. In this thesis, an enhanced version of CS-IIS is further proposed, in which all information to compute query-cluster similarities during query evaluation is stored. We introduce an incremental-CBR strategy that operates on top of this latter index structure, and demonstrate its search efficiency for different scenarios.

Finally, we exploit query views that are obtained from the search engine query logs to tailor more effective static pruning techniques. This is also related to the indexing task involved in a search engine. In particular, query view approach is incorporated into a set of existing pruning strategies, as well as some new variants proposed by us. We show that query view based strategies significantly outperform the existing approaches in terms of the query output quality, for both disjunctive and conjunctive evaluation of queries.

Keywords: Search engine, focused crawling, cluster-based retrieval, static index pruning.

ÖZET

ARAMA MOTORLARININ VERİMLİLİĞİNİ ARTIRMAK: ODAKLANMIŞ TARAMA, ARAMA VE İNDEKS BUDAMA STRATEJİLERİ

İsmail Sengör Altıngövde
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Özgür Ulusoy
Temmuz, 2009

Arama motorları, Ağ üzerinde bol miktarda bulunan metin verilerini getirmenin birincil aracıdırlar. Standart bir arama motoru üç temel görevi yerine getirir: Ağ tarama, indirilen içeriği indeksleme ve bu indeks üzerinde sorgu işleme. Bu işler için verimli yöntemler geliştirmek önemli bir araştırma konusudur. Bu tezde, bir arama motorunun yaptığı bu üç temel işe ilişkin verimli stratejiler önerilmektedir. Önerilen yöntemlerin çoğu, en geniş anlamıyla belge gruplarının (ki bunlar otomatik olarak elde edilmiş belge demetleri/sınıfları ya da elle düzenlenmiş kategorizasyonlar olabilir) halihazırda bulunduğu veya etkin bir şekilde elde edilebileceği durumlarda uygulanabilir. Ek olarak, sorgu görünümlerini kullanan bir statik indeks budama stratejisi de önerilmektedir.

Ağ tarama işi için, bir konu sınıflandırmasındaki belge sınıfları arasındaki kuralları kullanan kural-tabanlı bir odaklanmış tarama stratejisi önerilmiştir. Bu kurallar, iki sınıf arasındaki birbirlerine Ağ bağlantısı verme olasılığını temsil ederler. Önerilen kural-tabanlı tarayıcı, bir yol üzerindeki aranan konuya ilişkisiz sayfaları takip ederek konuyla ilişkili bir sayfaya ulaşabilmekte (yani *tünelleme* yapabilmekte) ve böylece aranan konuda daha yüksek oranda sayfa bulabilmektedir.

İndeksleme ve sorgu işleme kapsamındaysa belge gruplarını (demetler veya kategoriler) kullanarak arama yapma işine yoğunlaşılmıştır. Geleneksel demet-tabanlı getirme (DTG) senaryosunda, öncelikle verilen bir serbest metin sorgusuna en benzer belge demetleri belirlenir, sonra da bu demetlerdeki belgeler arasından sorgu yanıtı olanlar seçilip sıralanarak sunulur. Verimli DTG için, ilk olarak bazı alternatif sorgu işleme yöntemleri belirlenmiş ve değerlendirilmiştir.

Sonra, yeni bir indeks organizasyonu olarak demet-atlayan ters indeks yapısı (DA-TİY) tanıtılmıştır. Bu yeni yapıyı kullanan DTG'nin klasik indeks kullanan önceki stratejilere göre daha başarılı olduğu çeşitli veri kümeleri ve arama parametreleri kullanılarak gösterilmiştir. Bu tezde DA-TİY'in sorgu-demet benzerliğini hesaplamakta kullanılacak tüm bilgileri içeren daha geliştirilmiş bir hali de önerilmektedir. Bahsedilen indeks yapısı üzerinde çalışan artırımlı-DTG yaklaşımı tanıtılmakta ve farklı senaryolar için arama verimliliği gösterilmektedir.

Son olarak, arama motoru sorgu kütüklerinden elde edilen sorgu görünümleri kullanılarak daha başarılı statik indeks budama yöntemleri geliştirilmiştir. Bu da yine arama motorlarındaki indeksleme işiyle ilgilidir. Sorgu görünümü yaklaşımı literatürde bulunan çeşitli budama algoritmalarına ve bunların bizim tarafımızdan önerilen bazı başka biçimlerine yerleştirilmiştir. Sorgu görünümü tabanlı stratejilerin, mevcut diğer teknikleri hem "ve" hem de "veya" cinsi sorgu işleme durumlarında sorgu cevap kalitesi bakımından önemli ölçüde geçtiği gösterilmiştir.

Anahtar sözcükler: Arama moturu, odaklanmış tarama, demet-tabanlı getirme, statik indeks budama.

Acknowledgement

I would like to express my deepest thanks and gratitude to my supervisor Prof. Dr. Özgür Ulusoy for his invaluable suggestions, support, guidance and patience during this research.

I would like to thank all patient committee members for spending their time and effort to read and comment on my thesis. I owe special thanks to Prof. Dr. Fazlı Can for his never-ending encouragements and motivation. I inspired a lot from him. I am also indebted to Assoc. Prof. Dr. Uğur Güdükbay who did never give up giving me invaluable recommendations during this research. I would like to thank to Prof. Dr. Enis Çetin and Prof. Dr. Adnan Yazıcı for kindly accepting being in this committee.

Engin Demir, my nearest friend and colleague during the entire process of this research, deserves special thanks. Another friend, Berkant Barla Cambazoğlu, also deserves my gratitude for our never-ending technical (and less inspiring non-technical) discussions. I also thank to my colleague and office-mate Rıfat Özcan for his technical contributions and bearing to share the office with me.

I would like to thank to the former and current members of the Computer Engineering Department. I owe my warmest thanks to my friends Funda Durupınar, Meltem Çelebi, Deniz Üstebay, Duygu Atılgan, Kamer Kaya, Ediz Şaykol, A. Gökhan Aktürk, Ata Türk, Özlem Gür, Aylin Tokuç, Özlem N. Subakan-Yardibi, Derya Özkan, Nazlı İkizler-Cinbiş, Tayfun Küçükyılmaz, Cihan Öztürk, Selma Ayşe Özel and Yücel Saygın.

Finally, beyond and above all, I would like to thank my family.

To my family...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the digital age, data is abundant. The Web hosts an enormous amount of text data in various forms, such as Web pages, news archives, blogs, forums, manuals, digital libraries, academic publications, e-mail archives, court transcripts and medical records [122]. Search engines are the primary means of accessing the text content on the Web. To satisfy its users, a search engine should answer the user queries accurately and quickly. This is a demanding goal, which calls for a good and fast retrieval model and a large and up-to-date coverage of Web content.

To achieve these requirements, a search engine employs three main components [17, 34]: a crawler, to collect the Web resources; an indexer, to create an index of the text content, and a query processor, to evaluate the user queries. The first two tasks, crawling and indexing, are conducted off-line, whereas query processing is carried out on-line. Given the magnitude of the data on the Web, efficiency and scalability for each of these components are of crucial importance for the success of a search engine.

To have a better understanding of the requirements on search engines, let us

consider the growth of Web in the last decade. A major search engine, Google, announced the world's first billion-page index in 2000[1]. The number of indexed pages reached to 4.2 billion in 2004. At the time of this writing, major search engines, like Google and Yahoo!, are supposed to index approximately 40–60 billion pages. These figures imply a text collection and a corresponding index in the order of hundreds of terabytes, which can only be stored in clusters of tens of thousands of computers. Obviously, this evolution of Web data puts more pressure on satisfying the user needs; i.e., finding accurate results from the largest possible coverage of the Web, and doing it fast.

Subsequently, in the last two decades, a number of methods are proposed to improve the efficiency and scalability of a search engine. Paradigms from parallel and distributed processing are exploited for all components of a search engine, to cope with the growth of data. Furthermore, new approaches for crawling, indexing and query processing are introduced to improve the efficiency.

One such paradigm is prioritizing the Web pages and crawling only the "valuable" regions of Web, where the definition of a page's value depends on the specific application. Such *focused crawlers* cover only a specialized portion of Web and avoid the cost of crawling the entire Web, which is far beyond the capacity of individuals or institutions other than the largest players in the industry. The idea of focused crawling can be used to generate topical (also known as specialized, vertical, or niche) search engines that aim to provide high quality and up-to-date query results in a specific area for their users.

Such new approaches are also proposed for the other two components, namely indexer and query processor, of the search systems. For instance, a survey on inverted index files (i.e., the state-of-the-art index structure for large scale text retrieval) demonstrates that it is possible to significantly optimize these components by a number of techniques from recent research [122]. In comparison to a straightforward implementation, such techniques can reduce the disk space usage (up to a factor of five), memory space usage (up to a factor of twenty), query evaluation time in CPU (by a factor of three or more), disk traffic (by a factor of

[1] `http://www.google.com/corporate/history.html`

five in volume and two or more in time) and index construction time (by a factor of two).

Note that, the efficiency issues for search engines are also important to be able to provide higher quality results [34]. That is, devising efficient strategies for the components of a search engine may allow reserving more computing, storage and/or networking resources for improving the result quality. For instance, efficient crawling strategies may increase the coverage of the search engine, or efficient query processing strategies may allow more sophisticated ranking algorithms.

Given the key role of search engines for accessing information on the Web and the dynamicity, variability and growth of the Web data, exploring new methods for improving search efficiency is a popular topic that attracts many researchers from the academia and industry. In this thesis, we propose efficient strategies for the major components involved in a search engine.

1.2 Contributions

The contributions in this thesis consist of developing efficient techniques that are related to all three tasks, namely crawling, indexing and query processing, involved in a search engine. In the scope of the crawling task, we present a focused crawling strategy that exploits interclass rules. Next, we turn our attention to searching document groups; i.e., clusters and categories. To this end, we introduce a new inverted-index structure (and then, an enhanced version of it) and a cluster-based retrieval strategy. Clearly, these contributions are related to latter two tasks, namely; indexing and query processing, in a search engine. These strategies are essentially applicable when a grouping structure on top of the document collection is readily available or can be constructed in a feasible manner. That is, our approaches best fit to the cases where the collection is inherently clustered (e.g., as in a Web directory), or can be clustered/classified by an unsupervised clustering or supervised classification algorithm, respectively. Our work

includes several different scenarios corresponding to such cases. In this thesis, we also exploit search engine query logs for devising efficient methods for index pruning. In particular, we propose strategies using the query views for static index pruning. Our contributions in this context are most relevant to indexing task, as we present an off-line pruning approach for the underlying index. In the following paragraphs, we provide an overview of our particular contributions together with the organization of the thesis.

In Chapter 2 (based on [10]), we consider a focused crawling framework where Web pages are grouped (i.e., classified within a taxonomy) and the crawling is intended to find pages from a target class. We propose a rule-based focused crawling strategy that obtains and uses interclass rules while deciding the next page to be visited. These rules capture the probability of having hyperlinks between two classes. While crawling for a particular class, the rules are employed to assign higher priority to those pages that are from the target class or point to target class with high probability. We show that this strategy remedies some of the problems in a pioneering focused crawling strategy in the literature [48].

In Chapter 3 (based on [2, 3, 7, 37]), we again use document groups but for searching purposes. In particular, we consider both cases where documents are automatically clustered or manually categorized, and propose efficient strategies for typical cluster-based retrieval (typical-CBR). It is shown that, once the clusters that are most relevant to a query are obtained (or given by the users, as searching in Web directories [30, 31]), it is more efficient to use this information as early as possible while selecting the documents within from these clusters. As the major contribution of this chapter, we introduce a cluster-skipping inverted index structure (CS-IIS) and show that it is the most efficient approach for typical-CBR under realistic assumptions. We provide experimental evaluations using classical TREC [108] datasets that are automatically clustered in partitioning mode. Additionally, we discuss the use of typical-CBR strategy with CS-IIS in two different cases, namely, in a Turkish news portal and for searching within a hierarchical Web-directory.

In Chapter 4 (based on [4, 5]), we further enhance the CS-IIS discussed in

Chapter 3, and propose an incremental-CBR strategy, which interleaves selecting the clusters and documents that are most similar to a given query. We adapt state-of-the-art techniques for index compression and document identifier reassignment so that the storage requirements of the CS-IIS can be significantly reduced. We also show that our incremental-CBR strategy with CS-IIS can serve as a dynamic pruning approach in a framework in which Web pages are simply grouped according to their hosting Web sites.

In Chapter 5 (based on [8]), we propose exploiting query views to tailor more effective static index pruning strategies for both disjunctive and conjunctive query processing; i.e., the most common query processing modes in search engines. The query view approach is incorporated into a number of existing pruning techniques in the literature, as well as some adaptations proposed by us. An extensive comparison of all these techniques is also provided in a realistic experimental setup.

Finally, we conclude and point to some future work directions in Chapter 6.

Chapter 2

Exploiting Interclass Rules for Focused Crawling

A focused crawler is an agent that concentrates on a particular target topic and tries to visit and gather only relevant pages from a narrow Web segment. In this chapter, we exploit the relationships among document groups; more specifically, classes, to improve the performance of focused crawling. In particular, we extract rules that represent the linkage probabilities between different document classes and employ these rules to guide the focused crawling process.

In Sections 2.1 and 2.2, we provide a brief introduction to focused crawling and review the previous works in the literature, respectively. In Section 2.3, we first discuss the design issues for a general-purpose Web crawler. Then, we describe the baseline focused crawler, which is based on a pioneering work in the literature [48], and identify some problems of this approach. In Section 2.4, we introduce our rule-based focused crawling strategy. Experimental results for the proposed approach are presented in Section 2.5. Finally, we discuss our findings and point to future work in Section 2.6.

2.1 Introduction

With the very fast growth of WWW, the quest for locating the most relevant answers to users' information needs becomes more challenging. In addition to general purpose Web directories and search engines, several domain specific Web portals and search engines also exist, which essentially aim to cover a specific domain/topic (e.g., education), product/material (e.g., product search for shopping), region (e.g., transportation, hotels etc. at a particular country [33]) or media/file type (e.g., mp3 files or personal homepages [99]). Such specialized search tools may be constructed manually —by also benefiting from possible assistance of the domain experts— or automatically. Some examples of the automatic approaches simply rely on intelligent combination and ranking of results obtained from traditional search tools (just like meta search engines), whereas some others first attempt to gather the domain specific portion of the Web using focused crawling techniques and then apply other operations (e.g., information extraction, integration, etc.) on this collection.

To create a repository of Web resources on a particular topic, the first step is gathering (theoretically) *all* and *only* relevant Web pages to our topic of interest. A recently emerging solution for such a task is so-called *focused crawling* [45]. As introduced by Chakrabarti et al. [48], "A focused crawler seeks, acquires, indexes and maintains pages on a specific set of topics that represent a relatively narrow segment of the Web." Thus, an underlying paradigm for a focused crawler is implementing a *best-first search* strategy, rather than the *breadth-first search* applied by general-purpose crawlers.

In this chapter, we start with a focused-crawling approach introduced in [48] and use the underlying philosophy of their approach to construct a baseline focused crawler. This crawler employs a canonical topic taxonomy to train a naive-Bayesian classifier, which then helps determine the relevancy of crawled pages. The baseline crawling strategy also relies on the assumption of topical locality to decide which URLs to visit next. However, an important problem of this approach is its inability to support *tunneling*, i.e., it cannot tunnel toward the on-topic pages by following a path of off-topic pages [19]. To remedy this problem,

we introduce a rule-based strategy, which uses simple rules derived from interclass (topic) linkage patterns to decide its next move. Our experimental results show that the rule-based crawler improves the baseline focused crawler's harvest rate and coverage.

2.2 Related Work

A focused crawler searches the Web for the most relevant pages on a particular topic. Two key questions are how to decide whether a downloaded page is on-topic and how to choose the next page to visit [49]. Researchers have proposed several ideas to answer these two questions.

2.2.1 Early Algorithms

The FISHSEARCH system is one of the earliest approaches that has attempted to order the crawl frontier (for example, through a priority queue of URLs) [24]. The system is query driven. Starting from a set of seed pages, only those pages that have content matching a given query (expressed as a keyword query or a regular expression) and their neighborhoods (pages pointed to by these matched pages) are considered for crawling.

The SHARKSEARCH system [64] is an improvement over the former one. It uses a weighting method of term frequency (tf) and inverse document frequency (idf) along with the cosine measure to determine page relevance. SHARKSEARCH also smooths the depth cutoff method that its predecessor used.

Cho et al. [51] have also proposed reordering the crawl frontier according to page importance, which can be computed using various heuristics such as PageRank, number of pages pointing to a page (in-links), and so on. These early algorithms do not employ a classifier, but rather rely on techniques based on information retrieval (IR) to determine relevance.

2.2.2 Focused Crawling with Learners

Chakrabarti et al. [48] were the first to propose a soft-focus crawler, which obtains a given page's relevance score (i.e., relevance to the target topic) from a classifier and assigns this score to every URL extracted from that page. We refer to this soft-focus crawler as the *baseline focused crawler* and discuss in detail in Section 2.3.2. In a more recent work, they have proposed using a secondary classifier to refine the URL scores and increase the accuracy of this initial soft focused crawler [47]. This is also elaborated later in this chapter.

An essential weakness of the baseline focused crawler is its inability to model tunneling; that is, it cannot tunnel toward the on-topic pages by following a path of off-topic pages [19]. Two other remarkable projects, the context-graph-based crawler [56] and Cora's focused crawler [76], achieve tunneling.

The *context-graph* based crawler [56] also employs a best-search heuristic, but the classifiers used in this approach learn the layers which represent a set of pages that are at some distance to the pages in the target class (layer 0). More specifically, given a set of seeds, for each page in the seed set, pages that directly *refer to* this seed page (i.e., *parents* of the page) constitute layer-1 train set, pages that are referring to these layer-1 pages constitute the layer-2 train set, and so on; up to some predefined depth limit. The overall structure is called the context graph, and the classifiers are trained so that they assign a given page to one of these layers with a likelihood score. The crawler simply makes use of these classifier results and inserts URLs extracted from a *layer-i* page to the *layer-i* queue, i.e., it keeps a dedicated queue for each layer. URLs in each queue are also sorted according to the classifier's score. While deciding the next page to visit, the crawler prefers the pages nearest to the target class —that is, the URLs popped from the queue that correspond to the first nonempty layer with the smallest layer label. This approach clearly solves the problem of tunneling, but it requires constructing the context graph, which in turn requires finding pages with links to a particular page (*back links*). In contrast, our rule-based crawler uses forward links while generating the rules and transitively combines these rules to effectively imitate tunneling behavior.

CORA, on the other hand, is a domain-specific search engine on computer science research papers and it relies heavily on machine-learning techniques [76]. In particular, reinforcement learning is used in CORA's focused crawler. CORA's crawler basically searches for the expected future reward by pursuing a path starting from a particular URL. The training stage of classifier(s) involves learning the paths that may lead to on-topic pages in some number of steps. In contrast, our rule-based crawler does not need to see a path of links during training, but constructs the paths using the transitive combination and chaining of simple rules of length 1.

The focused crawler of Web Topic Management System (WTMS) fetches only pages that are close (i.e., parent, child, and sibling) to on-topic pages [80]. In WTMS, the relevancy of a page is determined by only using IR-based methods. In another work, Aggarwal et al. attempt to learn the Web's linkage structure to determine a page's likelihood of pointing to an on-topic page [1]. However, they do not consider interclass relationships in the way we do in this study. Bingo! is a focused-crawling system for overcoming the limitations of initial training by periodically retraining the classifier with high quality pages [103]. Recently, Menczer et al. present an evaluation framework for focused crawlers and introduce an evolutionary crawler [78]. In another work, Pant and Srinivasan provide a systematic comparison of classifiers employed for focused crawling task [86].

Two recent methods that exploit link context information are explored in [87]. In the first approach, so called text-window, only a number of words around each hyperlink is used for determining the priority of that link. The second one, tag-tree heuristic, uses the words that are in the document object model (DOM) tree immediately in the node that a link appears, or its parents, until a threshold is satisfied. In [6], we propose a similar but slightly different technique, so-called page segmentation method, which fragments a Web page according to the use of HTML tags.

Focused crawling paradigm is employed in a number of prototype systems for gathering topic/domain specific Web pages. For instance, in [106], focused

crawling is used for obtaining high quality pages on a mental health topic (depression). In [88], a prototype system is constructed that achieves focused crawling and multilingual information extraction on the laptop and job offers domains.

2.3 Baseline Focused Crawler

In this section, we first outline the design issues and architecture of a general-purpose crawler, based on the discussion in [45]. Next, we describe the focused crawler as proposed in [48], which is used as a baseline in our study.

2.3.1 A Typical Web Crawler

With the Web's emergence in the early 1990s, crawlers (also known as robots, spiders, or bots) appeared on the market with the purpose of fetching all pages on the Web; so that other useful tasks (such as indexing) can be done over these pages afterwards. Typically, a crawler begins with a set of given Web pages, called seeds, and follows all the hyperlinks it encounters along the way, to eventually traverse the entire Web [45]. General-purpose crawlers insert the URLs into a queue and visit them in a breadth-first manner. Of course, the expectation of fetching all pages is not realistic, given the Web's growth and refresh rates. A typical crawler runs endlessly in cycles to revisit the modified pages and access unseen content.

Figure 2.1 illustrates the simplified crawler architecture we implemented based on the architecture outlined in [45]. This figure also reveals various subtleties to consider in designing a crawler. These include caching and prefetching of Domain Name System (DNS) resolutions, multithreading, link extraction and normalization, conforming to robot exclusion protocol, eliminating seen URLs and content, and handling load balancing among servers (i.e., the politeness policy). We ignored some other issues, such as refresh rates, performance monitoring, and handling hidden Web, as they're not essential for our experimental setup.

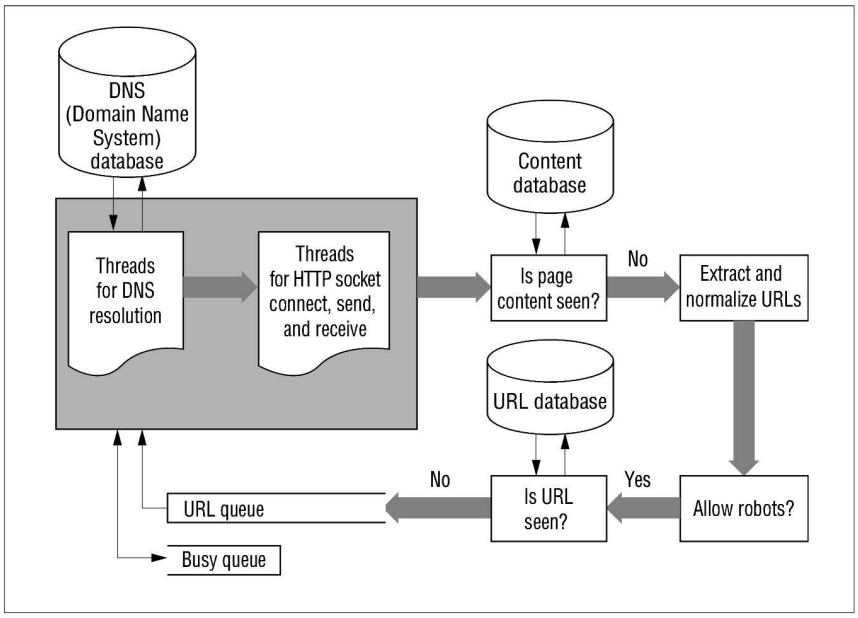


Figure 2.1: Our implementation of a typical crawler.

Our crawler operates as follows. The URL queue is initially filled with several seed URLs. Each DNS thread removes a URL from the queue and tries to resolve the host name to an Internet Protocol (IP) address. For efficiency purposes, a DNS database that basically serves as a cache is employed in the system. A DNS thread first consults the DNS database to see whether the host name has been resolved previously; if so, it retrieves the IP from the database. Otherwise, it obtains the IP from a DNS server. Next, a read thread receives the resolved IP address, tries to open an HTTP socket connection to the destination host, and asks for the Web page. After downloading the page, the crawler checks the page content to avoid duplicates. Next, it extracts and normalizes the URLs in the fetched page, verifies whether robots are allowed to crawl those URLs, and checks whether it has previously visited those extracted URLs (so that the crawler is not trapped in a cycle). For this latter purpose, the crawler hashes the URLs with the MD5 message-digest algorithm[1] and stores the hash values in the URL database. Finally, if this is the first time the crawler has encountered the URL, it inserts this URL into the URL queue; i.e., a first-in, first-out (FIFO) data structure in a general-purpose crawler. Of course, it is not desirable to overload

[1] `http://www.rsasecurity.com/rsalabs/node.asp?id=2253`

the servers with excessive number of simultaneous requests, so the first time the crawler accesses a server, it marks it as busy and stores it with a time stamp. The crawler accesses the other URLs from this server only after this time stamp is old enough (typically after a few seconds, to be on the safe side). During this time, if a thread gets a URL referring to such a busy server, the crawler places this URL in the busy queue. The threads alternate between accessing the URL queue and the busy queue, to prevent starvation in one of the queues. A more complicated solution, devoting a dedicated URL queue for each server, is left out for the purposes of this study. (For more details, see [45].)

2.3.2 Baseline Focused Crawler

On top of the basic crawler described in the above, we implemented the focused crawling strategy that is introduced in [48] as our baseline focused crawler (shortly, baseline crawler). We use this crawler to present our rule-based crawling strategy and to evaluate its performance. The baseline crawler uses a best-first search heuristic during the crawling process. In particular, both page content and link structure information are used while determining the promising URLs to visit.

The system includes a canonical topic (or, class[2]) taxonomy, i.e., a hierarchy of topics along with a set of example documents. Such a taxonomy can be obtained from the Open Directory Project[3] or Yahoo![4]. Users can determine the focus topics by browsing this taxonomy and marking one or more topics as the targets. In [48], it is assumed that the taxonomy induces a hierarchical partitioning of Web pages (i.e., each page belongs to only one topic), and we also rely on this assumption for our work.

An essential component of the focused crawler is a document classifier. In [48], an extended naive-Bayes classifier called *Rainbow* [77] is used to determine the crawled document's relevance to the target topic. During the training phase, this

[2]Note that, the terms "topic" and "class" are used interchangeably in this chapter.
[3]`www.dmoz.org`
[4]`www.yahoo.com`

classifier is trained with the example pages from the topic taxonomy, so that it learns to recognize the taxonomy.

Once the classifier constructs its internal model, it can determine a crawled page's topic; e.g., as the topic in the taxonomy that yields the highest probability score. Given a page, the classifier returns a sorted list of all class names and the page's relevance score to each class. Thus, the classifier is responsible for determining the on-topic Web pages. Additionally, it determines which URLs to follow next, assuming that a page's relevance can be an indicator of its neighbor's relevance; i.e., the *radius-1 hypothesis*. The radius-1 hypothesis contends that if page u is an on-topic example, and u links to v, then the probability that v is on-topic is higher than the probability that a randomly chosen Web page is on-topic [45].

Clearly, this hypothesis is the basis of the baseline focused crawler and can guide crawling in differing strictness levels [48]. In a *hard-focus* crawling approach, if the crawler identifies a downloaded page as off-topic, it does not visit the URLs found at that page; in other words, it prunes the crawl at this page. For example, if the highest-scoring class returned by the classifier for a particular page does not fall within the target topic, or if the score is less than a threshold (say, 0.5), the crawler concludes that this page is off-topic and stops following its links. This approach is rather restrictive with respective to its alternative, *soft-focus* crawling. In the latter approach, the crawler obtains from the classifier the given page's *relevance score* (a score on the page's relevance to the target topic) and assigns this score to every URL extracted from this particular page. Then, these URLs are inserted to a priority queue on the basis of these relevance scores. Clearly, the soft-focus crawler does not totally eliminate any pages but enforces a relevance-based prioritization among them. Another major component of the baseline crawler is the distiller, which exploits the link structure to further refine the URL frontier's ordering.

In our research, we did not include the distiller component in the baseline crawler implementation because we expect its effect to be the same for the baseline crawler and our rule-based crawler. In particular, our baseline focused crawler

includes a naive-Bayesian classifier and decides on the next URL to fetch according to the soft-focus crawling strategy. This means that in the architecture shown in Figure 2.1, we simply add a new stage immediately before the URL extraction stage to send the downloaded page to the classifier and obtain its relevance score to the target topic. We also replace the FIFO queues with priority queues.

2.4 Rule-Based Focused Crawler

An important problem of the baseline focused crawler is its inability to support tunneling. More specifically, the classifier employed in the crawler cannot learn that a path of off-topic pages can eventually lead to high-quality, on-topic pages. For example, if you're looking for neural network articles, you might find them by following links from a university's homepage to the computer science department's homepage and then to the researchers' pages, which might point to the actual articles (a similar example is also discussed by Diligenti et al. [56]). The baseline focused crawler described above would possibly attach low relevance scores to university homepages[5], which seems irrelevant to target topic of neural networks, and thus might miss future on-topic pages. The chance of learning or exploring such paths would further decrease as the lengths of the paths to be traversed increase.

As another issue, Chakrabarti et al. report that they have identified situations in which pages of a certain class refer not only to other pages of its own class (as envisioned by the radius-1 hypothesis) but also to pages from various other classes [48]. For example, they observed that pages for the topic "bicycle" also refer to "red-cross" and "first-aid" pages; and pages on "HIV/AIDS" usually refer to "hospital" pages more frequently than other "HIV/AIDS" pages. Such cases cannot be handled or exploited by the baseline crawler, as well.

To remedy these problems, we propose to extract rules that statistically capture linkage relationships among the classes (topics) and guide our focused crawler

[5]For instance, naive-Bayes classifiers are reported to be biased for returning either too high or too low relevance scores for a particular class [47].

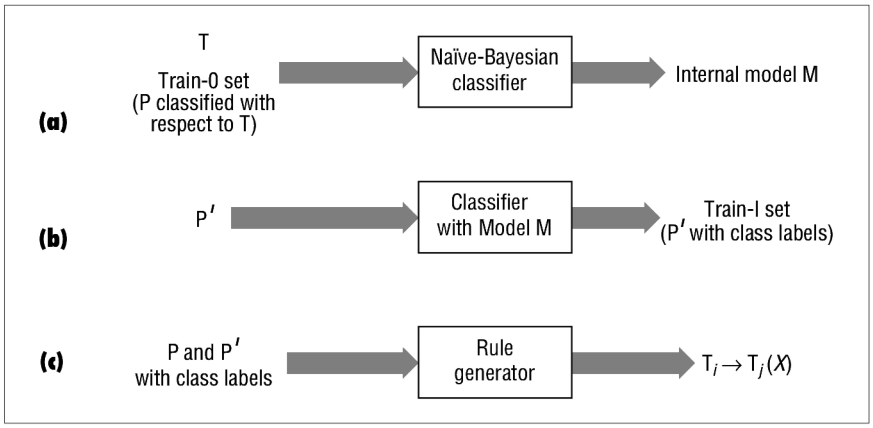


Figure 2.2: Stages of the rule generation process: (a) train the crawler's classifier with topic taxonomy T and the train-0 set to form internal model M, which learns T, (b) use page set P', pointed to by P, to form the train-1 set, (c) generate rules of the form $T_i \rightarrow T_j(X)$, where X is the probability score.

by using these rules. Our approach is based on determining relationships such as "pages in class A refer to pages in class B with probability p." During focused crawling, we ask the classifier to classify a particular page that has already been crawled. According to that page's class, we compute a score indicating the total probability of reaching the target topic from this particular page. Then, the crawler inserts the URLs extracted from this page into the priority queue with the computed score.

The training stage for our approach proceeds as follows. First, we train the crawler's classifier component with a class taxonomy and a set of example documents for each class, as in the baseline crawler. We call this the train-0 set (see Figure 2.2(a)). Next, for each class in the train-0 set, we gather all Web pages that the example pages in the corresponding class point to (through hyperlinks). Once again we have a collection of class names and a set of fetched pages for each class, but this time the class name is the class of parent pages in the train-0 set that point to these fetched documents. This latter collection is called the train-1 set. We give the train-1 set to the classifier to find each page's actual class labels (see Figure 2.2(b)). At this point, we know the class distribution of pages to which the documents in each train-0 set class point. So, for each class in the train-0 set, we count the number of referred classes in the corresponding train-1

Table 2.1: Class distribution of pages fetched into the train-1 set for each class in the train-0 set (for the example scenario)

Department homepages (DH)	Course homepages (CH)	Personal homepages (PH)	Sport pages (SP)
8 pages of class CH	2 pages of class DH	3 pages of class DH	10 pages of class SP
1 page of class PH	4 pages of class CH	4 pages of class CH	
1 page of class SP	4 pages of class PH	3 pages of class PH	

Table 2.2: Interclass rules of the example scenario for the distribution in Table 2.1 (the number following each rule is the probability score)

Department homepages (DH)	Course homepages (CH)	Personal homepages (PH)	Sport pages (SP)
DH → CH (0.8)	CH → DH (0.2)	PH → DH (0.3)	SP → SP (1.0)
DH → PH (0.1)	CH → CH (0.4)	PH → CH (0.4)	
DH → SP (0.1)	CH → PH (0.4)	PH → PH (0.3)	

page set and generate rules of the form $T_i \rightarrow T_j(X)$, meaning that a page of class T_i can point to a page of class T_j with probability score X (see Figure 2.2(c)). Probability score X is computed as the ratio of train-1 pages in class T_j to all pages in train-1 pages that the T_i pages in the train-0 set refer to. Once the rules are formed, they are used to guide the focused crawler. That is, a focused crawler seeking Web pages of class T_j would attach priority score X to the pages of class T_i that are encountered during the crawling phase.

To demonstrate our approach, we present an example scenario. Assume that our taxonomy includes four classes and a number of example pages for each class. The classes are "department homepages (DH)", "course homepages (CH)", "personal homepages (PH)" and "sports pages (SP)".

Next, for each class, we should retrieve the pages that this class's example pages refer to. Assume that we fetch 10 such pages for each class in the train-0 set and that the class distribution among these newly fetched pages (that is, the train-1 set) is as listed in Table 2.1. Then, the rules of Table 2.2 can be obtained in a straightforward manner.

Now, we compare the behavior of the baseline and rule-based crawlers to see how the rule-based crawler overcomes the aforementioned problems of the baseline crawler. Assume that we have the situation given in Figure 2.3(a). In this scenario, the seed page is of class PH, which is also the target class; that is, our crawler is looking for personal homepages. The seed page has four hyperlinks,

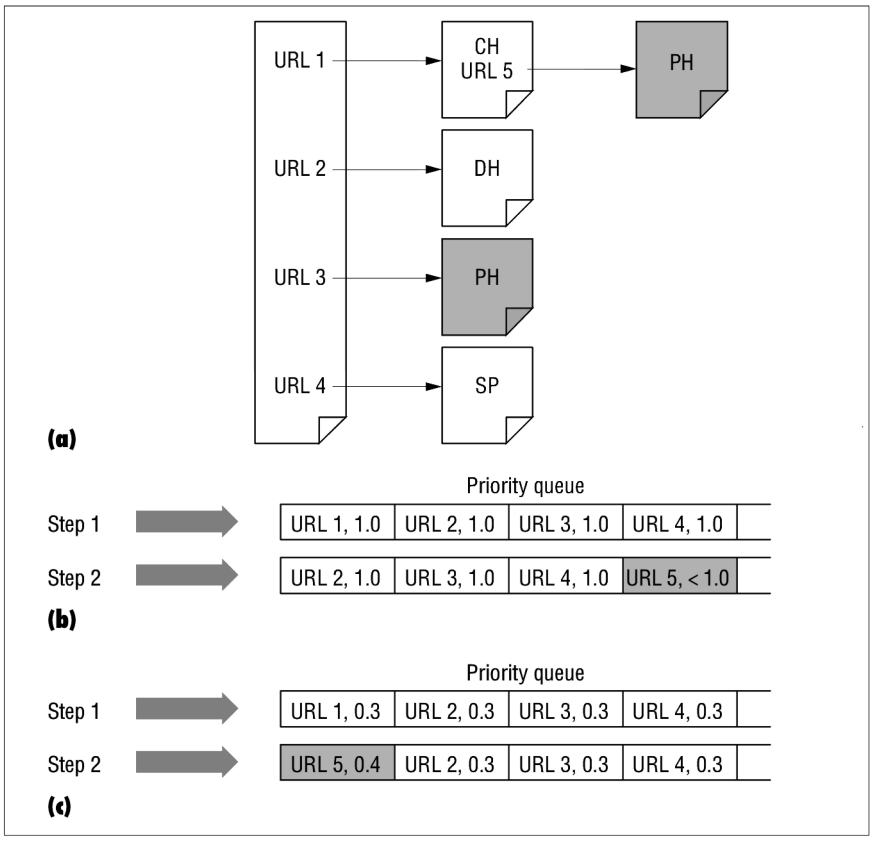


Figure 2.3: An example scenario: (a) seed page S of class PH, (b) steps of the baseline crawler, (c) steps of the rule-based crawler. Shading in (a) denotes pages from the target class; shading in (b) and (c) highlights where the two crawlers differ in Step 2.

such that links URL 1 through URL 4 refer to pages of classes CH, DH, PH, and SP, respectively. Furthermore, the CH page itself includes another hyperlink (URL 5) to a PH page.

As Figure 2.3(b) shows, the baseline crawler begins by fetching the seed page, extracting all four hyperlinks, and inserting them into the priority queue with the seed page's relevance score, which is 1.0 by definition (Step 1). Next, the crawler fetches URL 1 from the queue, downloads the corresponding page, and forwards it to the classifier. With the soft-focus strategy, the crawler uses the page's relevance score to the target topic according to the classifier. Intuitively, the CH page's score for target class PH would be less than 1, so the crawler adds URL 5, extracted from the CH page, to the end of the priority queue (Step

2). Thus, at best, after downloading all three pages with URLs 2 through 4, the crawler downloads the page pointed to by URL 5, which is indeed an on-topic page. If there are other intervening links or if the classifier score has been considerably low for URL 5, it might be buried so deep in the priority queue that it will never be recalled again.

In contrast, as Figure 2.3(c) shows, the rule based crawler discovers that the seed page of class PH can point to another PH page with probability 0.3 (due to the rules in Table 2.2), so it inserts all four URLs to the priority queue with score 0.3 (Step 1). Next, the crawler downloads the page pointed to by URL 1 and discovers that it is a CH page. By firing rule $CH \rightarrow PH(0.4)$, it inserts URL 5 to the priority queue, which is now at the head of the queue and will be downloaded next (Step 2), leading to an immediate award, i.e., an on-topic page. Figure 2.3 captures the overall scenario.

The rule-based crawler can also support tunneling for longer paths using a simple application of transitivity among the rules. For example, while evaluating URL 2 in the previous scenario, the crawler would learn (from the classifier) that the crawled page is of class DH. Then, the direct rule to use is $DH \rightarrow PH(0.1)$. Besides, the crawler can easily deduce that rules $DH \rightarrow CH(0.8)$ and $CH \rightarrow PH(0.4)$ exist and can then combine them to obtain path $DH \rightarrow CH \rightarrow PH$ with a score of $0.8 \times 0.4 = 0.32$ (assuming the independence of probabilities). In effect, the rule-based crawler becomes aware of path $DH \rightarrow CH \rightarrow PH$, even though it is trained only with paths of length 1. Thus, the crawler assigns a score of, say, the sum of the individual rule scores (0.42 for this example), to the URLs extracted from this DH and inserts these URLs into the priority queue accordingly.

Our rule-based scoring mechanism is not directly dependent of a page's similarity to the target page, but rather relies on the probability that a given page's class refers to the target class. In contrast, the baseline classifier would most probably score the similarity of a DH page to target topic PH significantly lower than 0.42 and might never reach a rewarding on-topic page.

2.4.1 Computing the Rule-Based Scores

There can be cases with no rules (for example, the train-0 and train-1 sets might not cover all possible situations). To handle such cases, the scoring mechanisms for soft-focus and rule-based crawling strategies can be simply combined. In Equation 2.1, we define the soft-focus strategy score as the likelihood of a page P being from class T, which is determined by the classifier model M [48].

$$S_{Soft} = M_{P,T} \tag{2.1}$$

Next, assuming independence of the probabilities, we define the score of a rule path R of $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k$, as in Equation 2.2, where $X_{i,j}$ denotes the probability score of the rule $T_i \rightarrow T_j$.

$$S_R = \prod_{i=1}^{k-1} X_{i,i+1} \tag{2.2}$$

Note that, as the rules can chain in a transitive manner, we define the MAXDEPTH as the maximum depth of allowed chaining. Typically, we allow rules to have a depth of at most 2 or 3. Also, when there is more than one path from an initial class to the target class, the crawler must merge their scores accordingly. Two potential merging functions are *maximum* and *sum*; and the latter is employed for the experiments reported in this work. The final scoring function of the rule-based crawling strategy for a URL u extracted from page P is given in Equation 2.3.

$$S_{P,u} = \begin{cases} \sum_{R \in RS} S_R, & \text{if } \exists \text{ rule path } R : T_i \rightarrow \cdots \rightarrow T_k \\ & \text{s.t. } length(R) < MAXDEPTH \text{ and } T_k \text{ is the target class;} \\ S_{Soft}, & \text{otherwise.} \end{cases} \tag{2.3}$$

where RS denotes a set of rule paths R, each of which has a length less than

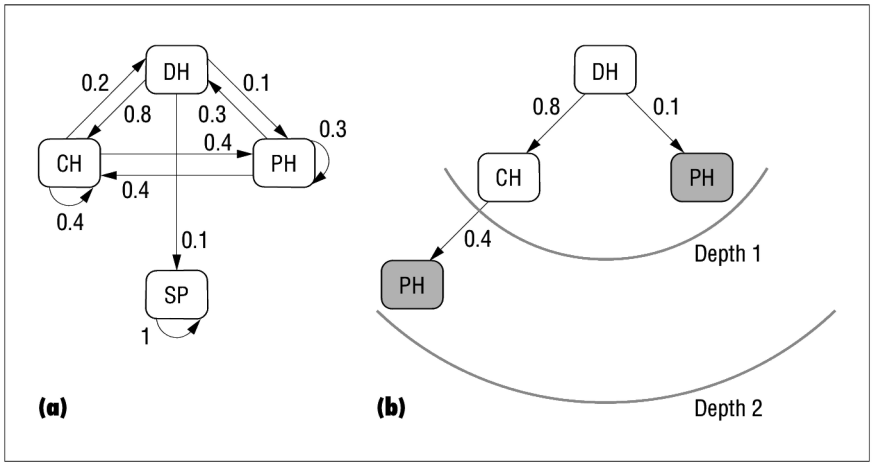


Figure 2.4: Rule-based score computation: (a) graph representation of the rule database and (b) computation of rule paths and scores –for example, a DH page has the score $(0.8 \times 0.4) + 0.1 = 0.42$. Once again, shading denotes pages from the target class.

MAXDEPTH and reaches to the target class, T_k.

Finally, all the rules and their scores for a particular set of target topics can be computed from the rule database before beginning the actual crawling. The rule database can be represented as a graph, as shown in Figure 2.4(a). Then, for a given a target class, T, the crawler can efficiently find all cycle-free paths that lead to this class (except the paths $T \rightarrow \cdots \rightarrow T$) by modifying the breadth-first-search algorithm (see [52] for a general discussion). For instance, in Figure 2.4(b) we demonstrate the rule-based score computation process for a page of class DH, where the target class is PH, and MAXDEPTH is 2.

2.5 Experiments

2.5.1 Experimental Setup

To evaluate our rule-based crawling strategy, we created the experimental setup described in earlier works [47, 48]. Train-0 set is created by using the ODP taxonomy and data [85], as follows. In ODP taxonomy, we moved the URLs found at a leaf node to its parent node if the number of URLs was less than a

predefined threshold (set to 150 for these experiments), and then we removed the leaf. Next, we used only the remaining leaves of the canonical class taxonomy; we discarded the tree's upper levels. This process generated 1,282 classes with approximately 150 URLs for each class. When we attempted to download all these URLs, we successfully fetched 119,000 pages (including 675,000 words), which constituted our train-0 set.

Next, due to time and resource limitations, we downloaded a limited number of URLs referred from the pages in 266 semantically interrelated classes on science, computers, and education in the train-0 set. This amounted to almost 40,000 pages, and constituted our train-1 set. Since the target topics for our evaluations are also chosen from these 266 classes, downloading the train-1 set sufficed to capture most of the important rules for these classes. We presume that even if we missed a rule, its score would be negligibly low (for example, a rule from `Top.Arts.Music.Styles.Opera` to `Top.Computer.OpenSource` might not have a high score, if it exists).

We employed the Bow library and the Rainbow text classifier as the default naive-Bayesian classifier [77]. We trained the classifier and created the model statistics with the train-0 data set in almost 15 minutes. Next, we classified the train-1 data set using the constructed model (which took about half an hour). In the end, using the train sets, we obtained 4,992 rules.

For crawling purposes, first a general purpose crawler (as shown in Figure 2.1) is implemented in C and then it is modified to support the baseline and rule-based focused crawling strategies described before. The underlying databases to store DNS resolutions, URLs, seen-page contents, hosts, extracted rules and the URL priority queues are all implemented by using Berkeley DB[6]. During the crawling experiments, the crawler is executed with 10 DNS and 50 read threads.

[6] `www.sleepycat.com`

2.5.2 Results

Here, we provide performance results for three focused crawling tasks using the baseline crawler with the soft-focus crawling strategy and our rule-based crawling strategy. The target topics were `Top.Science.Physics.QuantumMechanics`, `Top.Computer.History`, and `Top.Computer.OpenSource`, for which there were 41, 148, and 212 rules, respectively, in our rule database. For each topic, we constructed two disjoint seed sets of 10 URLs each, from the pages listed at corresponding entries of the ODP and Yahoo! directories.

The performance of a focused crawler can be evaluated with the harvest ratio, a simple measure of the average relevance of all crawler-acquired pages to the target topic [45, 48]. Clearly, the best way to solicit such relevance scores is to ask human experts; however, this is impractical for crawls ranging from thousands to millions of pages. So, following the approach of earlier works [48], we again use our classifier to determine the crawled pages' relevance scores. The harvest ratio is computed as in Equation 2.4.

$$HR = \frac{\sum_{i=1}^{N} Relevance(URL_i, T)}{N} \tag{2.4}$$

In Equation 2.4, $Relevance(URL_i, T)$ is the relevance of the page (with URL_i) to target topic T as returned by the classifier, and N is the total number of pages crawled.

Table 2.3 lists the harvest ratio of the baseline and rule-based crawlers for the first few thousands of pages for each target topic and seed set. The harvest ratios vary among the different topics and seed sets, possibly because of the linkage density of pages under a particular topic or the quality of seed sets. The results show that our rule-based crawler outperforms the baseline crawler by approximately 3 to 38 percent. Second, we provide the URL overlap ratio between the two crawlers. Interestingly, although both crawlers achieve comparable harvest ratios, the URLs they fetched differed significantly, implying that the coverage of

Table 2.3: Comparison of the baseline and rule-based crawlers; percentage improvements are given in the column "impr."

Target topic	Evaluation metrics	Seed set 1			Seed set 2		
		Baseline	Rule based	Impr. (%)	Baseline	Rule based	Impr. (%)
Quantum mechanics	Harvest ratio	0.28	0.30	7.1	0.25	0.29	16
	URL overlap	10%	10%	NA	16%	16%	NA
	Exclusive HR	0.27	0.29	7.4	0.23	0.28	22
Computer history	Harvest ratio	0.29	0.40	38.0	0.36	0.37	3
	URL overlap	27%	27%	NA	22%	22%	NA
	Exclusive HR	0.26	0.39	50.0	0.35	0.37	6
Open source	Harvest ratio	0.52	0.56	9.0	0.48	0.61	27
	URL overlap	10%	10%	NA	8%	8%	NA
	Exclusive HR	0.51	0.54	6.0	0.47	0.61	30

these crawlers also differs. For each crawler, we extracted the pages exclusively crawled by it and computed the harvest ratio. The last row of Table 2.3 for each topic shows that the harvest ratio for pages that the rule-based crawler exclusively crawled is also higher than the harvest ratio for pages that the baseline crawler exclusively crawled.

In our second experiment, we investigate the effects of seed set size on crawler performance. To this end, we searched Google with the keywords "open" and "source" and used the top 50 URLs to constitute a seed set. The harvest ratios were similar to the corresponding case with the first seed set in Table 2.3. For this case, we plot the harvest rate, which is obtained by computing the harvest ratio as the number of downloaded URLs increased. The graph in Figure 2.5 reveals that both crawlers successfully keep retrieving relevant pages, but the rule-based crawler does better than the baseline crawler after the first few hundred pages.

2.6 Conclusions and Future Work

In this study, we propose and evaluate an intuitive rule-based approach to assist guiding a focused crawler. Our findings are encouraging in that the rule-based crawling technique achieves better harvest ratio with respect to a baseline classifier while covering different paths on the Web. In a recent paper [47], Chakrabarti et al. enhanced the baseline crawler with an *apprentice*, a secondary classifier that

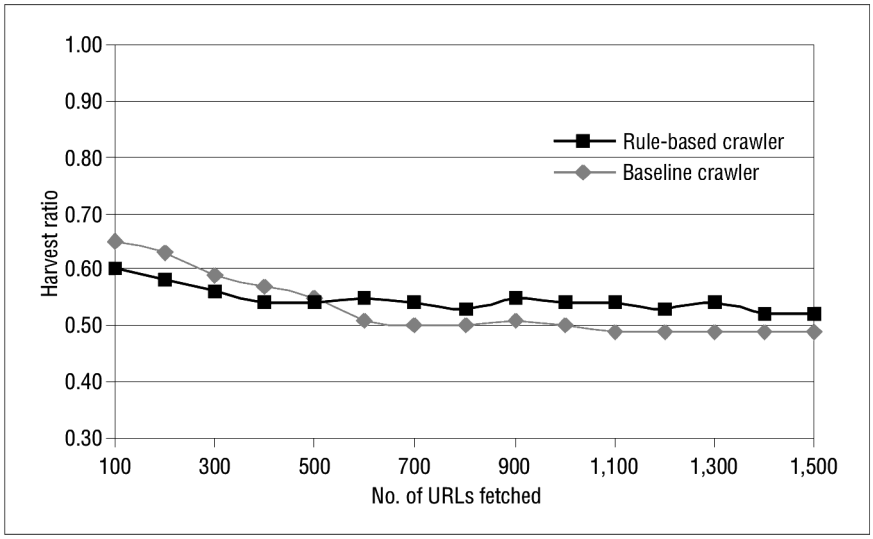


Figure 2.5: Harvest rates of baseline and rule-based crawlers for the target topic `Top.Computer.OpenSource`, 50 seeds.

further refines URL ordering in the priority queue. In one experiment, they provided their secondary classifier with the class name of a page from which the URL was extracted, which resulted in a up to 2% increase in accuracy. They also report that because of a crawler's fluctuating behavior, it is difficult to measure the actual benefit of such approaches. We experienced the same problems, and in the future, we plan to conduct further experiments to provide more detailed measurements.

Chakrabarti and his colleagues also investigated the structure of broad topics on the Web [46]. One result of their research was a so-called topic-citation matrix, which closely resembles our interclass rules. However, the former uses sampling with random walk techniques to determine the source and target pages while filling the matrix, whereas we begin with a class taxonomy and simply follow the first-level links to determine the rules. Their work also states that the topic-citation matrix might enhance focused crawling. It would be interesting and useful to compare and perhaps combine their approach with ours.

Our research has benefited from earlier studies [45, 48], but it has significant differences in both the rule generation and combination process as well as in the

computation of final rule scores. Nevertheless, considering the diversity of the Web pages and topics, it is hard to imagine that a single technique would be the most appropriate for all focused-crawling tasks. Our experimental results also justify this claim and are promising for future research.

Our rule-based framework can be enhanced in several ways. It is possible to employ more sophisticated rule discovery techniques (such as the topic citation matrix we have discussed), refine the rule database online, and consider the entire topic taxonomy instead of solely using the leaf level.

Chapter 3

Search Using Document Groups: Typical Cluster-Based Retrieval

In the following two chapters of the thesis, we concentrate on the efficiency of search using document clusters. Typical cluster-based retrieval (CBR) is a two stage process where for a given free-text query first the *best-clusters* that are most relevant to the query are selected and then the *best-documents* that match the query are determined from within these clusters. In this chapter our goal is providing efficient means for typical-CBR. To this end, we first propose different query processing strategies and then introduce a new index organization; namely, cluster-skipping inverted index structure (CS-IIS). Finally, we provide extensive experimental evaluation of the proposed strategies using automatically clustered and manually categorized datasets and for automatically or manually determined best-clusters sets.

The rest of this chapter is organized as follows. In Section 3.1, we provide the motivation for our work. In Section 3.2, we provide a review of query processing in large scale IR systems, giving more emphasis to cluster-based retrieval studies in the literature. Section 3.3 proposes alternative query processing strategies for typical-CBR. In Section 3.4, we introduce CS-IIS to be used for CBR. The following two sections, 3.5 and 3.6, are devoted respectively to the experimental setup and results where the proposed approaches are evaluated on TREC datasets that are automatically clustered by a partitioning algorithm. In Section 3.7, we provide further results, but this time using the largest Turkish corpora in

the literature. Finally, in Section 3.8, we evaluate the success of typical-CBR with CS-IIS for the case of searching Web directories that involve a hierarchical clustering of documents. Conclusive remarks are given in Section 3.9.

3.1 Introduction

In an information retrieval (IR) system the ranking-queries, or Web-like queries, are based on a list of terms that describe user's information need. Search engines provide a ranked document list according to potential relevance of documents to user queries. In ranking-queries, each document is assigned a matching score according to its similarity to the query using the vector space model [96]. In this model, the documents in the collection and queries are represented by vectors, of which dimensions correspond to the terms in the vocabulary of the collection. The value of a vector entry can be determined by one of the several term weighting methods proposed in the literature [97]. During query evaluation, query vectors are matched with document vectors by using a similarity function. The documents in the collection are then ranked in the decreasing order of their similarity to the query and the ones with highest scores are returned. Note that Web search engines exploit the hyperlink structure of the Web or the popularity of a page for improved results [25, 74].

However, exploiting the fact that document vectors are usually very sparse, an inverted index file can be employed instead of full vector comparison during the ranking-query evaluation. Using an inverted index, the similarities of those documents that have at least one term in common with the query are computed. Throughout this thesis, a ranking-query evaluation with an inverted index is referred to as full search (FS). Many state-of-the art large-scale IR systems such as Web search engines employ inverted files and highly optimized strategies for ranking-query evaluation [122].

An alternative method of document retrieval is first clustering the documents in the collection into groups according to their similarity to each other. Clusters are represented with centroids, which can include all or some of the terms

that appear in the cluster members. During query processing, only those clusters that are most similar to the query are considered for further comparisons with cluster members; i.e., documents. This strategy, so-called cluster-based retrieval (CBR) is intended to improve both efficiency and effectiveness of the document retrieval systems [67, 95, 98, 113]. It can improve efficiency, as the query-document matches are computed for only those documents that are in the clusters most similar to the query. Furthermore, it may enhance effectiveness, according to the well-known cluster hypothesis [110, 111]. Note that, the resulting ranking returned by CBR can be different from that of FS, as the former considers only those documents in the promising clusters.

Surprisingly, despite these premises of CBR for improving effectiveness and efficiency, the information retrieval community has witnessed contradictory results in terms of both aspects in the last few decades [73, 96, 113]. This inconsistency relatively reduced the interest on CBR and its consideration as an alternative retrieval method to full search. On the other hand, the growth of Web as an enormous digital repository of every kind of media, and essentially text, also creates new opportunities for the use of clustering and CBR. For example, Web directories (e.g., ODP [85], Yahoo!, etc.), a major competitor of search engines, allow users browse through the categories and assign a query on a particular category. This is a kind of CBR, except that clusters are browsed manually. Furthermore, there exist several large-scale text repositories that are available on Web or on proprietary networks with again manual and/or automatic classification/clustering of the content. Clearly, CBR, as a model of information retrieval, perfectly fits to the requirements of such environments, given that the suspects on its effectiveness and efficiency are remedied. A recent attempt addressing the effectiveness front is by Liu and Croft [73], which shows that by using language models CBR effectiveness can be significantly better than what it is assumed to be in the literature. The efficiency of CBR is investigated in Chapters 3 and 4 of this thesis.

For any given IR system involving document clusters (or categories) -created either automatically or manually, for legacy data or Web documents and in a flat

or hierarchical structure- the best-match CBR strategy has two stages: i) best(-matching) clusters selection: the clusters that are most similar to the submitted query are determined by using cluster centroids; ii) best(-matching) documents selection: the documents from these best-matching clusters are matched with the query to obtain the final query result. In the early days of IR, once best-clusters are obtained, it is presumed to be a reasonable strategy to compare the query with the document vectors of the members of those clusters (exhaustive search). This may be a valid and efficient strategy if the clusters are rather small and queries are rather long. In contrary, the state-of-the art applications for CBR, such as Web directories or digital libraries, involve collections with large number of documents with respect to number of clusters (or, categories) and attempt to respond a very high load of typically short queries. Indeed, the inefficiency of the exhaustive strategy has been long recognized [96, 113]. As a remedy, the use of inverted index files for both stages of CBR (i.e., comparison with centroids and documents) has been proposed [36] (please see Section 3.2.3.3 for a more detailed discussion). More specifically, once the best-clusters are obtained, a full search is conducted over the entire collection to find the documents that have non-zero similarity to the query; i.e., the candidates to be the best-documents. Next, among these documents, only those from the best-clusters are filtered to be presented to the user. This is a practical approach that is also applied for Web directories [30, 31]. In this work, we refer to this strategy using an IIS for both stages as *typical-CBR*.

However, typical-CBR still involves some significant redundancy. At the best-documents selection stage, the inverted index is used to find "all" documents that have non-zero similarity to the query (note that, this is nothing but the FS). Since only documents from best-clusters are returned, the computations (decoding the postings, computing partial similarities, updating accumulators, inserting into and extracting from the heap for the final output, etc.) for the eliminated documents are all wasted. Furthermore, there is the cost of computing best-clusters. If the index files are kept on disk (a relaxable assumption considering the advances in the hardware, as we discuss later), accessing these structures requires

two direct (random) disk accesses per query term, one for the centroid and another for document posting lists (assuming that the lists are read entirely once located on disk). These issues imply that, typical-CBR, as defined here; cannot be a competitor of FS in terms of efficiency, as it already involves the cost of FS in addition to the latter costs specific to best-clusters selection stage.

In Chapters 3 and 4, we attempt to remedy above problems and devise more efficient means of searching clusters. In this chapter, we first explore how the performance of typical-CBR can be further improved by using the best-cluster information as early as possible during query processing and propose alternative strategies. The major contribution of this chapter is a new data structure, so-called cluster-skipping inverted index structure (CS-IIS) that blends cluster membership information and typical postings. We show that, typical-CBR using CS-IIS outperforms the other CBR strategies, and even FS, given that the best-cluster selection cost is excluded. This latter case is possible when best-clusters are provided by the users, say, by browsing in a Web directory. In Chapter 4, we will further relax this condition and introduce a new CBR strategy that can be as efficient as FS, even when best-clusters have to be computed automatically for a query. Finally, our experiments in this chapter are in an environment where the files are not compressed, whereas evaluation with compression is reported in Chapter 4.

3.2 Related Work and Background

In the following, we first review the two basic IR strategies, namely FS and typical-CBR, and their implementations employing an IIS for ranking-queries. For the sake of completeness, we also include inverted index compression and optimization techniques for FS in this section, although these latter issues are discussed in Chapter 4.

3.2.1 Full Search using Inverted Index Structure (IIS)

In an IR system, typically two basic types of queries are provided: Boolean and ranking-queries. In the former case, query terms are logically connected by the operators AND, OR and NOT and those documents that make this logical expression true (i.e., satisfy the query) are retrieved. In ranking-queries (or, free-text queries [75]), each document is assigned a matching score according to its similarity to the query using the vector space model [98]. In this thesis, we concentrate on the ranking-queries, which are more frequently used in the Web search engines and IR systems. However, our approach proposed here can be applicable to Boolean queries, as well.

In the vector space model, documents of a collection are represented by vectors. For a document collection including T distinct terms, each document is represented by a T-dimensional vector. For those terms that do not appear in the document, the corresponding vector entries are zero. On the other hand, the entries for those terms that appear in the document can be determined by one of the several "term weighting" methods described in the literature [97]. The goal of these methods is to assign higher weights to the terms that can potentially discriminate a document among others, and vice versa. One of the most widely used weighting methods is the *term frequency (tf)* $\times$ *inverse document frequency (idf)* formulation. While computing the weight of term t in document d, denoted as $w_{d,t}$, *tf* is computed as the number of occurrences of t in d, and *idf* is $\ln(\frac{number\ of\ documents}{number\ of\ documents\ including\ t}+1)$. In the literature, several variants of *tf-idf* scheme are proposed (such as sublinear *tf* scaling or augmented *tf* normalization, see [75] for a general discussion). For example, augmented normalized frequency formula for a term t in document d is defined as $0.5 + (0.5 \times f_{d,t})/max\text{-}tf$. Here *max-tf* denotes the maximum number of times any term appears in d. The term weights for query terms $w_{Q,t}$ can also be calculated in a similar fashion to document term weights.

After obtaining weighted document (d) and query (Q) vectors in a T dimensional vector space the query-document matching is performed using the, so-called, cosine similarity function [98] shown in Equation 3.1.

$$Similarity(Q,d) = \sum_{t \in Q} w_{Q,t} \times w_{d,t} \tag{3.1}$$

Note that, for Equation 3.1 we assume that a term's weight in a document is computed by using the *tf-idf* formula, and then *normalized* by using the document length. Document lengths (denoted as W_d) are computed using Equation 3.2. No normalization is needed for query terms since it does not affect document ranking.

$$W_d = \sqrt{\sum_{t \in d} (w_{d,t})^2} \tag{3.2}$$

There are other weighting methods and similarity functions based on statistical principles (such as the well known Okapi BM25 metric) or language models. These methods also make use of the term frequencies, document lengths, etc. but in a different manner. A detailed discussion of these methods are available in [75, 118, 122].

It is possible to evaluate ranking-queries very efficiently by using an inverted index of document vectors. In this case, the query vector is not matched against every document vector (most of which would probably yield no similarity at all), but only those that have at least one common term with the query. Indeed, we can safely state that an inverted index is the state-of-the-art data structure for processing ranking-queries in large scale IR systems and Web search engines. An inverted file has a header (also called as vocabulary) part, including list of terms in the collection, and pointers to the posting lists for each term. Along with the terms, f_t, number of documents in which this term appears, is kept. A posting list for a term consists of the documents that include the term and is usually a list of (document id d, within-document term frequency $f_{d,t}$) pairs. The posting lists are stored contiguously on disk. This is usually called a *document-level* index and adequate to process Boolean and ranking queries [122]. It is also possible to capture the positions of each term within the document in the postings, to be able process phrase and proximity queries. In the rest of this thesis, all ordinary inverted index structures keep only document identifiers and term frequency information; i.e., they are document-level, unless explicitly stated

otherwise.

During ranking-query evaluation, an accumulator structure with as many entries as the collection size is kept in the memory (note that variations are possible [61, 122]). The weighted query vector is constructed as described above. For each term t in the query vector Q, a direct access is made to the disk to locate t's posting list by using the pointer stored in the IIS header. Once located, the posting list associated with this term t is read sequentially (as it is stored on contiguous disk blocks) and brought to main memory. For each document d in the posting list, first $w_{d,t}$ is computed by using the *tf-idf* formula (or, some other weighting method). Note that, the *tf* component corresponds to the $f_{d,t}$ values that are stored along with the document ids in the posting lists. The *idf* component can be easily computed using term frequency f_t stored in the IIS header. Next, using a similarity function the partial similarity of the query to the document is computed (i.e., $w_{d,t} \times w_{Q,t}$ for the cosine function [118]) for this particular term, and the resulting value is added to the accumulator entry for this document. After all query terms are processed in the same manner, the entries of accumulator are normalized; i.e., divided by the pre-computed document lengths. Finally, the accumulators (documents) are sorted in descending similarity order and returned as the query output. If only top-k documents are required and k is much smaller than the collection size, which is the common case as in the Web, using the *min-heap* data structure significantly reduces the query processing time. Further details of ranking-query processing are discussed extensively in [32, 34, 118, 122].

In this study, a ranking-query evaluation as described in the previous paragraph is referred to as full search (FS). It is "full" in the sense that it returns exactly the same results as the sequential collection scan and uses all terms in the documents (except stop words, as we mention in the experimental setup). The query evaluation algorithm for FS is given in Algorithm 1 (based on [118, 122]). Note that, the details of query-document partial similarity computations and length normalization are not shown, as they are dependent on the actual term weighting and scoring function.

Algorithm 1 Typical ranking-query evaluation algorithm for free-text queries

Input: Query Q, Index I
Output: Top-k best matching documents
1: **for** each term t in Q **do**
2: Retrieve I_t from I
3: **for** each posting $(d, f_{d,t})$ in I_t **do**
4: $DAcc[d] \leftarrow DAcc[d] + \text{PartialSimilarity}(d, Q)$
5: Build a min-heap H of size k for nonzero $DAcc$ entries
6: Extract top-k best-matching documents from H

3.2.1.1 Compression of IIS

There are several works regarding the compression of inverted indexes, and in this section we briefly summarize them based on the discussion in [118]. The key point for compressing posting lists is storing the document ids in list elements as a sequence of *d-gaps*. For instance, assume that posting list for a term t includes the following documents: 3, 7, 11, 14, 21, 24; using d-gaps this can be stored as 3, 4, 4, 3, 7, 3. In this representation, the first document id is stored as-is whereas all others are represented with a d-gap (id difference) from the previous document id in the list. The expectation is that the d-gaps are much smaller than the actual ids. Among many possibilities, variable-length encoding schemes are usually preferred to encode d-gaps and term frequencies as they allow representing smaller integers in less space than larger ones. There are several bitwise encoding schemes. In the next chapter, we will focus on the Elias-γ and Golomb codes, following the approach implemented in [79, 118]. More recently, Anh and Moffat [13] propose another compression scheme, which is also applicable in our framework.

In the literature, a particular choice for encoding typical posting list elements (i.e., $(d, f_{d,t})$ pairs) is using the Golomb and Elias-γ schemes for d-gaps and term frequency values, respectively [118]. Elias-γ code is a non-parameterized technique that allows easy encoding and decoding. Golomb code is a parameterized technique, which, for some parameter b, encodes a nonzero integer x in two parts. For inverted index compression, the parameter b can be determined by using a global Bernoulli process modeling the probabilistic distribution of document id

occurrences in posting lists. Golomb code can be further specialized by using a local Bernoulli model for each posting list. In this case, the d-gaps for frequent terms (with longer posting lists) are coded with small values of b, whereas d-gaps for less frequent terms are coded with larger values. During encoding and decoding, the b value is determined for a particular posting list I_t by Equation 3.3.

$$b = 0.69 \times \frac{N}{f_t} \tag{3.3}$$

where N is the number of documents, and f_t is the frequency of term t in the collection (i.e., the length of the posting list I_t).

3.2.1.2 Optimization Techniques for FS

There are various optimization techniques used for inverted index searches [12, 14, 15, 26, 27, 71, 79, 89, 90]. These techniques aim to use only the most promising parts of posting lists and try to increase efficiency of query processing without deteriorating retrieval effectiveness. For instance, *quit* and *continue* techniques enforce a limit on the number of accumulator entries that can be updated during query evaluation. In this case, memory consumption is reduced as the accumulators for storing partial similarities can be implemented by dynamic data structures instead of a collection-size array. Furthermore, these two strategies coupled with a skipping index are shown to improve Boolean and ranking-query efficiency [79]. Persin et al. propose to use frequency-sorted indexes to avoid reading entire posting lists from the disk [90]. More recently, Anh et al. introduced impact-sorted lists to improve the efficiency of FS [12, 15].

Using skip-elements in an inverted file to improve query evaluation efficiency in a non-clustering environment was first proposed in [79]. They compress posting lists by using some fixed length skips, which serve as synchronization points, and are able to decompress posting lists from any point of skips without decompressing the undesired parts. For example, the (posting) list (1, 5, 10, 13, 18, 23, 50, 57, 58, 60) could be reorganized with four synchronization points: 1, 13, 50, and 60. For simplicity let us assume that we are in a conjunctive Boolean query environment, and also assume that another list has already been processed and

it is known that the query has no answers between documents 13 and 50, then the original list only needs to be accessed (and decompressed) up to document 13 and after document 50 —this means 7 posting list positions instead of 10.

Our cluster-skipping inverted file proposed in this thesis is inspired by this former work, but extends it in various ways. We leave the details of this approach to Section 4.5.2.3 of Chapter 4, where we also provide an experimental comparison to our approach.

3.2.2 Document Clustering for IR

Clustering algorithms group a set of documents into subsets, or clusters [75]. They essentially fall into two groups: partitioning and hierarchic. The partitioning algorithms, such as K-means and C^3M, produce a flat clustering of documents, which may or may not belong to more than one clusters; whereas hierarchic ones yield a hierarchy of clusters. The hierarchic algorithms are either top-down or bottom-up. The bottom-up approach, also known as hierarchical agglomerative clustering (HAC), starts with individual documents and then proceeds by successively merging pairs of documents or clusters. The most widely implemented HAC algorithms are single-link, average-link, complete-link and Ward's algorithm.

A good survey of clustering in information retrieval is provided in [116]. The books by Salton [95, 96], Salton and McGill [98], van Rijsbergen [110] and Anderberg [11] also cover previous work on clustering in information retrieval. A more recent survey of clustering in various application areas can be found in [66]. A good discussion of algorithms for clustering data and cluster validation approaches is available in a beautiful concise book by Jain and Dubes [65]. In this thesis, without loss of generality, we use a partitioning algorithm, C^3M, to automatically cluster the document collections. The details of this algorithm are discussed in the next section.

There are various applications of clustering in IR as we briefly summarize from [75]. A recent use of clustering is in result-presentation (for instance, see [107]) where a clustered set of search results are presented to a user, instead

of a plain top-k list. This approach is also applied successfully in a commercial search engine, *Vivisimo*[1]. In another application, so-called scatter-gather, clustering is used as a means of building a better search interface [62]. Liu and Croft revisit clustering for improving retrieval effectiveness while using language models. Finally, a classical use of clustering is for improving retrieval efficiency; i.e., cluster-based retrieval [50, 94, 95, 111]. This latter usage is the core of Chapters 3 and 4 of this thesis, and thus elaborated in more depth in Section 3.2.3.

3.2.2.1 C^3M Clustering Algorithm

In the experimental evaluations of this work, we use the C^3M algorithm, which is known to have good information retrieval performance. The C^3M algorithm assumes that the operational environment is based on the vector space model. Using this model, a document collection can be abstracted by a document, D, matrix of size m by n whose individual entries, $d_{i,j}(1 \leq i \leq m, 1 \leq j \leq n)$, indicate the number of occurrences of term j (t_j) in document i (d_i).

Determining the number of clusters in a collection is a difficult problem [65]. In other clustering algorithms, if it is required, the number of clusters, n_c, is usually a user specified parameter; in C^3M it is determined by using the cover-coefficient (CC) concept [42, 120, pp. 376-377]. In C^3M some of the documents are selected as cluster seeds and non-seed documents are assigned to one of the clusters initiated by the seed documents. According to CC, for an m by n document matrix the value range of n_c and the average cluster size (d_c) are as follows.

$$1 \leq n_c \leq min(m, n);\ max(1, m/n) \leq d_c \leq m$$

In C^3M, the document matrix D is mapped into an m by m cover-coefficient (C) matrix using a double-stage probability experiment. This asymmetric C matrix shows the relationships among the documents of a database. Note, however, that the implementation of C^3M does not require the complete C matrix. The diagonal entries of C are used to find the number of clusters, n_c, and the selection of cluster seeds. During the construction of clusters, the relationships between a

[1] `http://vivisimo.com/`

non-seed document (d_i) and a seed document (d_j) is determined by calculating the $c_{i,j}$ entry of C, where $c_{i,j}$ indicates the extent with which (d_i) is covered by (d_j). Therefore, the whole clustering process implies the calculation of ($m+$ ($m-$ n_c) $\times$ n_c) entries of the total m^2 entries of C. This is a small fraction of m^2, as $n_c \ll m$ (for some examples please refer to Table 3.1 in Section 3.5). A thorough discussion and complexity analysis of C^3M are available in [42].

The CC concept reveals the relationships between indexing and clustering [42]. These relationships can be used to predict the clustering structure generated by the algorithm. The CC-based indexing-clustering relationships are formulated as follows.

$n_c = t/(x_d \times t_g) = (m \times n)/t = m/t_g = n/x_d$, and $d_c = m/n_c = t_g$

In these formulas, the meanings of the variables not used in the text so far are as follows.

$d_c : m/n_c$, average number of documents per cluster,

t: total number of non-zero entries in D matrix,

$t_g : t/n$, average number of different documents a term appears (term generality), and

$x_d : t/m$, average number of distinct terms per document (depth of indexing).

It is shown that the algorithm can be used in a dynamic environment in an incremental fashion and such an approach saves clustering time and generates a clustering structure comparable to that of cluster regeneration by C^3M [35, 38]. C^3M and its concepts have also attracted the attention of other researchers in various application areas, such as chemical information systems [41, 117], clustering tendency testing [57], automatic hypertext structure generation [68], and search output clustering [70].

3.2.3 Search Using the Document Clusters

The well-known clustering hypothesis states that "closely associated documents tend to be relevant to the same request." It is this hypothesis that motivates clustering of documents in a collection [110]. In the IR research, automatic clustering of documents has been originally introduced with the expectation of increasing the efficiency and effectiveness of the retrieval process [67, 95]. The premise of CBR is restricting the search to only the most-relevant clusters that is determined by the query-cluster similarity, and thus improving the efficiency.

Today, CBR is not only limited to automatically clustered collections. In the last two decades, we witnessed the creation of some of the largest document hierarchies, Web directories (such as Yahoo! and ODP [85]) that attracts a certain attention of users. Such directories allow the users to browse through categories or issue queries that are restricted to a certain subset of these categories [30, 31]. This is again a form of CBR, where the clusters to be searched are explicitly determined by the user.

In what follows, we first discuss representation of clusters from the retrieval point of view. Then, we briefly discuss some of the earlier CBR approaches that are essentially addressing the collections with hierarchical clusters. Finally, we discuss more recent techniques that employ inverted index files during CBR.

3.2.3.1 Cluster Centroids

A classical issue for the cluster-based retrieval is deciding on the terms that should appear in the cluster representatives; i.e., centroids, and determining the maximum centroid vector length and centroid term weights. Murray [82] states that the effectiveness of retrieval does not increase linearly with the centroid length. Thus, in the literature, a limited number of terms selected by various methods are used as cluster centroids. For instance, in hierarchical clustering experiments described by Voorhees [112, 113], the sum of $f_{d,t}$ values of each term in a cluster is computed and the terms are sorted by decreasing frequency. Next, top-k terms are selected as the cluster centroid, where an appropriate value

of k is experimentally determined [112]. Note that, based on Murray's centroid definition [82], Voorhees attempted to find the shortest centroid vectors that cause minimal deterioration on effectiveness. However, the results reported in that work show variability to draw a conclusion for the relationship of the centroid length and effectiveness for several hierarchical CBR techniques. This earlier work uses centroid size of 250 while noting that "further research into a theory of centroid creation and weighting" is required.

Several other methods are also proposed for selecting centroid terms. In [67], terms that appear in more than $\log_2(C)$ documents, where C is the cluster size, are selected as this cluster's centroid terms. Yet another approach may be selecting terms that have a total $f_{d,t}$ value greater than the average of $f_{d,t}$ values for the terms in the cluster. Muresan and Harper [81] propose to use cluster terms that have positive Kullback-Leibler divergence score. A recent work [107], which reviews many of these methods for deriving cluster centroids, claims that "the effect of centroids on CBR effectiveness has not been extensively investigated" and "the challenge raised by Voorhees seventeen years ago still stands unaddressed". In this study, we employ several centroid selection and weighting methods and compare their impact on retrieval effectiveness and efficiency.

3.2.3.2 Earlier CBR Strategies with Vector Comparisons

In a hierarchical clustering setup [112, 113], a CBR system requires several files: the representation of the cluster hierarchy, the centroid vectors and the document vectors. In this setup, a top-down search begins by placing the root of the cluster hierarchy into a max-heap [118]. During the search the top element of the heap, which has the highest similarity to the query, is extracted. If it is a document, it is added to the output set. If the extracted element is a cluster, then its children, which may be other clusters or documents, are inserted into the heap according to their similarity to the query (only those with non-zero similarity are considered). The top-down search ends when the heap is empty or a pre-defined number of documents is retrieved. Notice that, the actual centroid and document vectors — but not index files— are employed during the query-cluster and query-document

similarity computations.

A bottom up search strategy, again for hierarchical environments, starts with the top ranking document(s) that is at the bottom of the cluster tree, and goes up looking for proper clusters. This approach needs the top ranking document(s) information, which can only be obtained by a full search (the study also introduces another method that uses the centroids of bottom-most clusters) [53]. The search may switch back and forth between documents and clusters.

3.2.3.3 Typical-CBR using the Inverted Index Structure (IIS)

In partitioning clustering, a flat clustering of the documents is produced and the search is typically achieved by the best-match strategy. The best-match CBR search strategy has two stages i) selection of n_s number of best-clusters using centroids, ii) selection of d_s number of best-documents of the selected best-clusters. For item (i) we have two file structure possibilities: centroid vectors and IIS of centroids. For item (ii) we again have two possibilities: document vectors and IIS of all documents. One remaining possibility for (ii), a separate inverted index for the members of each cluster, is not considered due to its excessive cost in terms of disk accesses (for a query Q with q terms it would involve q direct disk accesses for each selected cluster) and maintenance overhead. Hence, possible combinations of (i) and (ii) determine four different implementation alternatives.

In [36] the efficiency of the above alternatives is measured in terms of CPU time, disk accesses, and storage requirements in a simulated environment defined in [113]. It is observed that the alternative employing an IIS for both centroids and documents (separately) is significantly better than the others. Notice that, the query processing in this case is quite similar to ranking-query evaluation for FS discussed in Section 3.2.1, and repeats this procedure twice, using centroid IIS and document IIS, respectively. A final stage is also required for filtering those documents that are retrieved by the second stage (i.e., FS using the document index) but do not belong to the best-clusters. A similar approach is typically used for processing queries restricted to certain categories on Web directories (with the only distinction that best cluster(s) are explicitly specified by the user instead

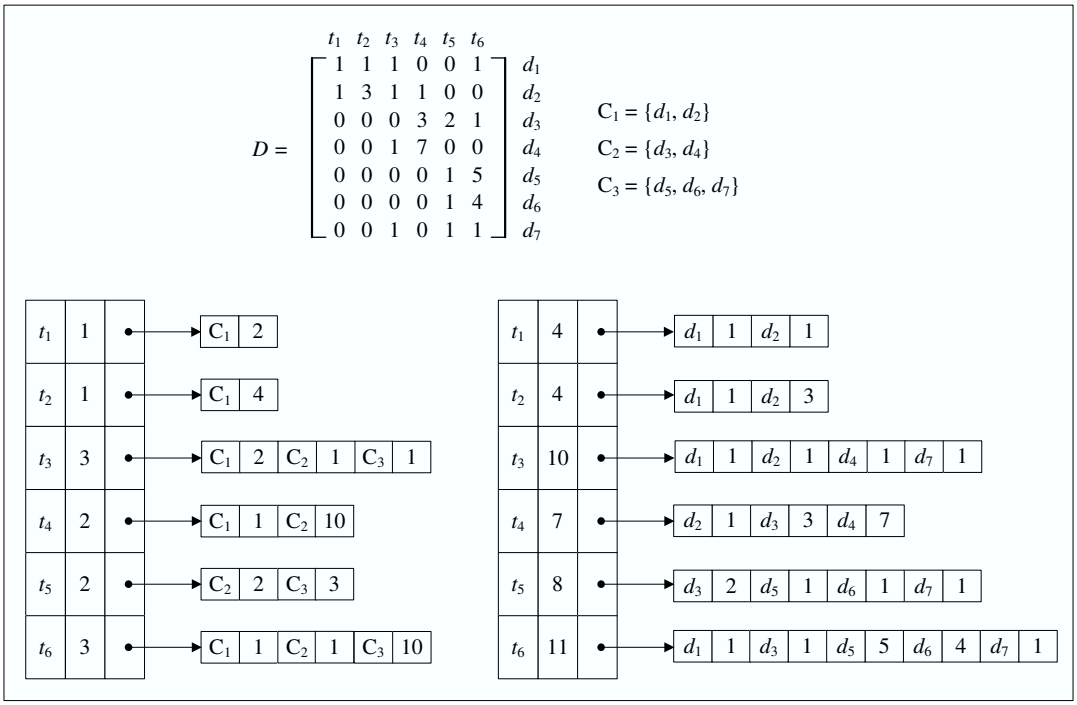


Figure 3.1: Centroid and document IIS for typical-CBR.

of an automatic computation) [30, 31]. Throughout the thesis, we refer to this particular two-stage approach using an IIS for each stage as *typical-CBR*.

Example 3.1 In Figure 3.1, we illustrate the centroid and document IIS files for this strategy. The example provided in Figure 3.1 is for a document by term D matrix with three clusters C_1, C_2, and C_3. In the D matrix, rows and columns respectively indicate documents and terms. It shows that document 2 (d_2) contains term 1 (t_1) once and t_2 three times. We assume that, for simplicity, all terms appearing in the member documents of a cluster are used in the centroid and the centroid inverted index is created accordingly. For instance, term t_1 appears in two documents, d_1 and d_2, once in each. Since both documents are in C_1, the posting element for C_1 in the list of t_1 stores the value 2 as the within-cluster term frequency (i.e., $f_{C,t}$).

In [36], it is further stated that typical-CBR is inferior to FS in terms of query evaluation efficiency. This is an expected result, as the best-document selection stage of typical-CBR is actually nothing but a full search on the entire

collection. Furthermore, selecting the best-clusters and the final result filtering would also incur additional costs. This latter cost, integrating the best-clusters and documents is discussed in detail below.

3.2.3.4 Result Integration Stage for Typical-CBR

Once the best-clusters and best-documents are obtained separately using corresponding inverted index files, there are two ways to eliminate the best-documents that are not a member of the best-clusters [30, 31]; i.e., to integrate the results of best-cluster and document selection stages. We call these alternatives "document-id intersection based integration" and "cluster-id intersection based integration", and describe in detail next.

- *Document-id intersection based integration:* This alternative uses an inverted index such that for each cluster, the documents that fall into this particular cluster are stored (i.e., cluster-document (CD)-IIS). In this case, by using this latter index, first the union of all documents that are within the best-clusters is determined, and then the resulting document set is intersected with the best-documents to obtain the final result. Note that, in an IR environment with clustering, such an inverted index of documents per cluster (i.e., a member document list for each cluster) is required in any case, to allow the browsing functionality.

- *Cluster-id intersection based integration:* The second integration alternative is just the reverse: for each document in the best-document set, the cluster(s) in which this document lies is found by using an (inverted) index that stores the list of clusters for each document (i.e., document-cluster (DC)-IIS). Then, the obtained cluster id(s) are intersected with the best-clusters set and if the result is not empty, the document is added to the final query output set.

The first integration alternative would be efficient when the number of documents per cluster is relatively small, whereas the second approach would be more

efficient when the best-document set to be processed is small. Also note that, the inverted index required by the second alternative is redundant, as it is the transpose of the CD-IIS that would be implemented in any case to support the browsing functionality. On the other hand, as the integration process required by the first alternative requires first obtaining a union of several document lists and then an intersection, it would be less efficient in terms of query processing time, whereas storing an additional inverted index (DC-IIS) is not a major concern given the storage capabilities of modern systems [30]. In this thesis, we assume that the cluster-id intersection based integration, which seems to be more practical for large-scale IR systems, is employed in typical-CBR. In Section 3.3, we propose alternative query processing strategies for typical-CBR under this assumption and discuss their efficiency trade-offs.

Example 3.2 Consider the centroid and document IIS files in Figure 3.1. Let us assume that the user query Q contains the terms $\{t_3, t_5\}$, $n_s = 1$ (i.e., a single best-cluster would be selected) and d_s= 2 (i.e., top-2 documents will be retrieved). At the first stage, the query is forwarded to the centroids IIS and postings list for each term is processed. Since C_3 has the highest total term frequency in these lists, let us assume that it is determined as the best-cluster. Next, the query is sent to document IIS. Notice that the postings for the query terms include all documents d_i (for $1 \leq i \leq 7$), so all of these documents would be ranked as they have non-zero similarity to query. Assume they are ranked as from d_1 to d_7, for simplicity. For each of these documents, its cluster id would be intersected with the best-clusters set. Considering the cluster information in Figure 3.1, only documents d_5, d_6 and d_7 are from the best-clusters set, which includes C_3 in our case; and top-2 of these documents, d_5 and d_6, will be retrieved as the query output. Note that, in practice, it would suffice to rank a reasonably larger number of documents than d_s, instead of ranking all documents that have non-zero similarity to the query. This issue is further elaborated in Section 3.3.

3.2.3.5 Typical-CBR using Modified Inverted Index Structures

To avoid the integration step mentioned above, it is proposed to modify the inverted index used during the document selection. In [31], document identifiers in the postings are created as signatures, which convey information about the hierarchy of clusters in which a document belongs to. However, since the signatures can produce false drops; i.e., only provide an approximate filtering of best-document set, there is still a need for the cluster-id based integration approach to obtain the final query result. In [30], another, so-called, optimistic approach is introduced, which embeds the cluster information into the actual document identifiers in the index. That is, for a, say, 32-bit document identifier, the first 10-bits are used to represent the cluster identifier of the document, whereas the rest of the bits store the actual document identifier. Note that, this structure requires bitwise processing of identifiers during query evaluation. In Section 3.4, we propose a new, cluster-skipping inverted index structure that both eliminates the result integration stage and improves the performance of the best-document selection stage.

3.3 Query Processing Strategies for Typical-CBR

As discussed in the previous section, typical-CBR involves selection of best-clusters and documents, which is followed by a result integration stage. Let us assume that the best-clusters set is already obtained either automatically (i.e., by query-cluster matching using the centroid IIS) or manually (i.e., by browsing, as in a category-restricted query [30, 31]). Then, a typical ranking query evaluation algorithm as shown in Algorithm 1 can be employed during the best-document selection stage. Finally, those documents that are not from the best-clusters can be discarded from the query output. In this section, we propose different strategies for the best-document selection stage so that result integration can be achieved earlier during the processing. We show that these strategies improve the performance under certain conditions.

The query processing strategies discussed here differ in how they answer the following questions: (i) at what point during the best-document selection should the cluster-id(s) of a particular document be intersected with the best-cluster ids, and (ii) what kind of data structure should be used to keep best-cluster ids? Considering the query evaluation shown in Algorithm 1, the cluster ids can be intersected at three different points, yielding three implementation alternatives: (i) before updating the accumulator entry for a document, (ii) before inserting a document to the min-heap, or (iii) after extracting the top scoring documents from the min-heap (i.e., the traditional baseline approach as described in [30, 31]). Two potential data structures to store best-cluster ids are (i) a sorted array of best-clusters, or (ii) a 0/1 mark array in which entries for best-clusters are 1 and all others are 0. We discuss these alternatives and their efficiency trade-offs in the following.

Intersect Before Update (IBU). In this approach (Algorithm 2), only those accumulator entries that belong to documents from best-clusters are updated. To achieve this, after a posting list is retrieved for a query term, the cluster to which each document in the posting list belongs is determined and intersected with the best-cluster set. If the document's cluster is found in the best-cluster set, its accumulator entry is updated. Otherwise, there is no need to compute the partial query-document similarity and accumulator update for this particular document.

Note that, this alternative would also increase the efficiency of the last two steps of the algorithm (i.e., building and extracting from the heap as shown in lines 7-8 in Algorithm 2), since all of the nonzero entries in the accumulator structure are for the documents that are from best-clusters. On the other hand, the performance of this approach crucially depends on the cost of determining the clusters to which a document belongs (line 4) and cluster-id intersection operation (line 5). For the former operation, the algorithm should access document-cluster (DC) IIS for each element of the posting lists. However, if document-cluster associations are kept in the main memory or cached efficiently, this cost can be avoidable. This seems reasonable, since DC-IIS can be expected to be relatively small in size and can be shared among several query processing threads. For

Algorithm 2 The query processing algorithm for intersect before update (IBU) approach

Input: Query Q, Index I, Best-clusters $BestClus$, Document-category index I_{DC}
Output: Top-k best matching documents

```
1: for each term t in Q do
2:     Retrieve I_t from I
3:     for each posting (d, f_{d,t}) in I_t do
4:         Retrieve I_d from I_DC
5:         if I_d ∩ BestClus ≠ ∅ then
6:             DAcc[d] ← DAcc[d] + PartialSimilarity(d, Q)
7: Build a min-heap H of size k for nonzero DAcc entries
8: Extract top-k best-matching documents from H
```

instance, assuming that documents are not repeated in more than one clusters, the main memory requirement to cache the entire DC-IIS would be $O(N)$, i.e., in the order of the number of documents. In this study, without loss of generality, we assume that each document belongs to at most one cluster and the DC-IIS is stored in the main memory.

Assuming each document belongs to only one cluster, the cost of a cluster-id intersection is $O(\log S)$, if a sorted array of size S is used to store best-cluster ids; and $O(1)$ if a 0/1 mark array is used for this purpose. Note that, the data structure for best-clusters can be a sorted array if the memory reserved per query is scarce and/or total number of clusters is quite large. In this case, the document's cluster id can be searched within best-clusters using binary search. A 0/1 mark array is obviously more efficient but can only be preferred if the memory is not a concern and/or number of clusters is relatively small. Finally, if the number of best-clusters is relatively small, which is possible in a practical setup, a hash-table can also be used instead of a mark array to provide similar look-up efficiency but less space consumption.

Intersect Before Insert (IBI). In this approach, instead of applying the cluster id intersection for each doc-id in each posting list, we do it once for each non-zero accumulator entry while building the heap (Algorithm 3). This alternative is preferable if the number of non-zero accumulator entries is expected to be low and/or the cost of cluster id intersection is high, e.g., DC-IIS is on disk.

Algorithm 3 The query processing algorithm for intersect before insert (IBI) approach

Input: Query Q, Index I, Best-clusters $BestClus$, Document-category index I_{DC}
Output: Top-k best matching documents

```
1: for each term t in Q do
2:     Retrieve I_t from I
3:     for each posting (d, f_{d,t}) in I_t do
4:         DAcc[d] ← DAcc[d] + PartialSimilarity(d, Q)
5: for each Dacc[d] ≠ 0 do
6:     Retrieve I_d from I_DC
7:     if I_d ∩ BestClus ≠ ∅ then
8:         Insert d into the min-heap H of size k
9: Extract top-k best-matching documents from H
```

Intersect After Extract (IAE). As illustrated in the example in Section 3.2.3.4, this is the simplest result integration approach that is probably employed in current systems (e.g., [30, 31]). Roughly, in this approach the best-document selection stage proceeds as FS, and the elimination of documents that are not from best-clusters are achieved at the very end. This approach allows an existing IR system using FS to easily adapt a clustering or classification structure on top of its document collection without any modification; but, in turn, cannot utilize the best-clusters information while selecting best-documents. We still outline this strategy for the sake of completeness and to use it as a baseline in the evaluation of strategies that we propose above and in the next section.

In this strategy (Algorithm 4) the entire query processing works as in Algorithm 1 and only at the end of the evaluation, the cluster-ids of top-k documents are intersected with the best-clusters. Of course, if some of those k documents are not from the best-clusters, then the build-heap step and extraction should be repeated. To avoid such a repetition, the initial evaluation can be executed for top-L documents, where $L > k$. In this case, the cost of cluster-id intersection would be negligible as it is postponed at the end of processing and $L \ll N$. On the other hand, it is important to choose L appropriately, if L is much larger than k (e.g., $L = N$ as an extreme case), the gains in the intersection stage would be lost during the build-heap and extraction. If L is too small (i.e., very close to k), we may need more than one iteration to find at least k documents that are in the best-clusters. Thus, IAE alternative will be useful if it can somehow be

Algorithm 4 The query processing algorithm for intersect after extract (IAE) approach

Input: Query Q, Index I, Best-clusters $BestClus$, Document-category index I_{DC}
Output: Top-k best matching documents

```
1: for each term t in Q do
2:     Retrieve I_t from I
3:     for each posting (d, f_{d,t}) in I_t do
4:         DAcc[d] ← DAcc[d] + PartialSimilarity(d, Q)
5: Build a min-heap H of size L(L ≥ k) for nonzero DAcc entries
6: ResultNum ← 0
7: while ResultNum < k and H is not empty do
8:     Extract d with the highest score from H
9:     Retrieve I_d from I_DC
10:    if I_d ∩ BestClus ≠ ∅ then
11:        Insert d into output, ResultNum ← ResultNum + 1
12: if ResultNum < k then
13:    Set L to some M s.t. M > L, go to Line 5
```

guaranteed that in a small number of highest scoring documents, there will be at least k documents from the best-clusters. More specifically, this approach would be better than the previous alternative only if cluster intersection is costly; and better than the IBU algorithm if both intersection test is expensive and too many nonzero accumulator entries arise.

3.4 Cluster-Skipping Inverted Index Structure for Typical-CBR

In the previous section, we described some strategies for implementing the best-document selection and result integration stages in a more efficient manner. In this section we introduce a new inverted index organization that has the potential of further improving the CBR efficiency. In this data structure, the *(document, term frequency)* pairs in a posting list are re-organized such that all documents from the same cluster are grouped together, and at the beginning of each such group an extra element, so-called skip-element, is stored in the form of *(cluster id, next cluster address)*. During the best-document selection stage, if the cluster id in that additional index element is not found in the best-cluster set, the documents in that cluster are skipped and the query processor jumps to the next cluster

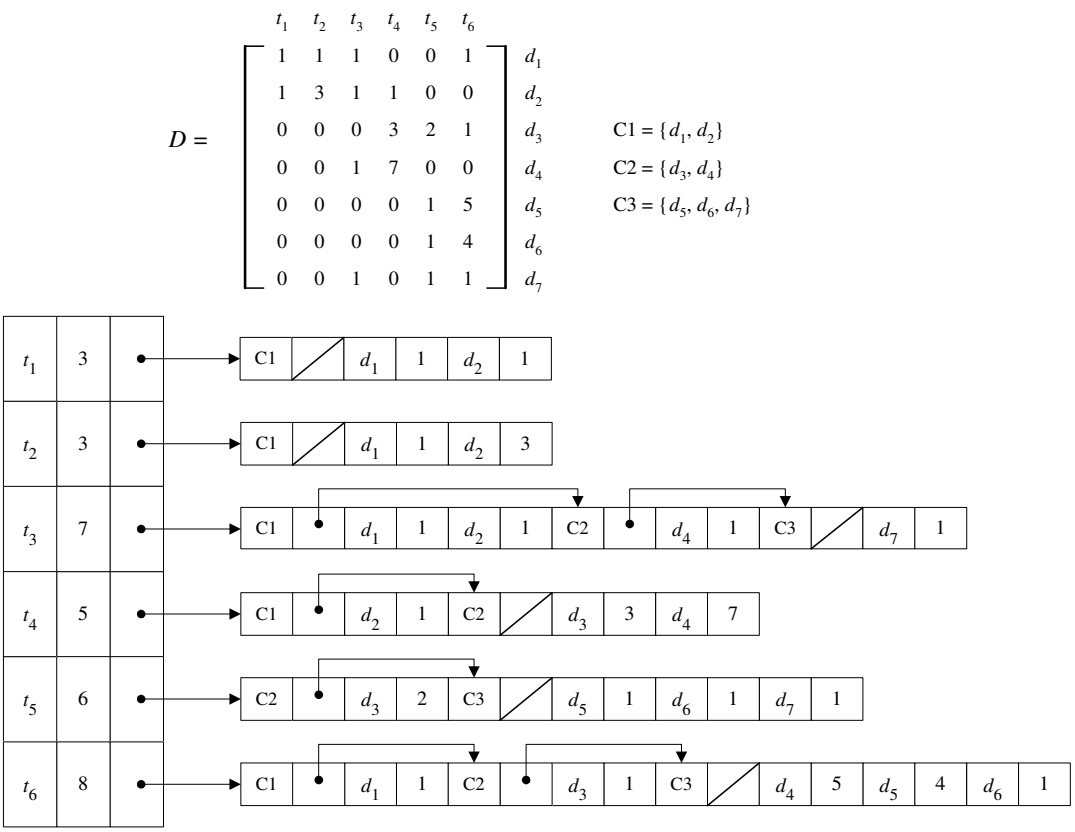


Figure 3.2: Cluster-skipping inverted index structure (CS-IIS) for typical-CBR.

pointed by the "next cluster address". Thus, for each posting list, only the parts that include documents from the best-clusters are processed.

An example file structure for our approach is provided in Figure 3.2 for a D matrix. In this figure each posting list header contains the associated term, the number of posting list elements associated with that term, and the posting list pointer (disk address). The posting list elements are of two types, *(cluster id, next cluster address)* and *(document id, term frequency)* for the preceding cluster.

Our skip structure is simple yet novel. In the previous CBR research a similar approach has not been used. For example, Salton and McGill's classical textbook [98, pp. 223-224] defines three cluster search strategies. Two of them are related to hierarchical cluster search and their concern is the storage organization of the cluster centroids. In the third CBR strategy, documents (not their

inverted lists) are stored in cluster order, that is, one access to the "document file" retrieves a cluster of related documents. Our skip idea provides a completely new way of implementing CBR by clustering the individual posting lists elements. This is certainly different than accessing the "documents" in cluster order.

In [96, p. 344] Salton states that: "In general, the efficiencies of inverted-file search techniques are difficult to match with any other file-search system because the only documents directly handled in the inverted-list approach are those included in certain inverted lists that are known in advance to have at least one term in common with the queries. In a clustered organization, on the other hand, many cluster centroids, and ultimately many documents, must be compared with query formulations that may have little in common with the queries."

The CBR using the skip-based inverted index search technique overcomes the problem stated by Salton; i.e., it prevents matching many unnecessary documents with the queries. For example, in the clustering environment of Figure 3.2, if we assume that the user query contains the terms $\{t_3, t_5\}$ and the best-clusters for this query are $\{C_1, C_3\}$, using the CS-IIS during query processing after selecting the best-clusters we only consider the posting lists associated with t_3 and t_5. While processing the posting list of t_3 we skip the portion corresponding to C_2 (since it is not a best-cluster). Similarly, while processing the posting list of t_5, we again skip the unnecessary C_2 portion of the posting list and only consider the part corresponding to C_3. In other words, by using the skip approach we only handle the documents that we really need to match with the query. Note that, during query processing, best-clusters can be stored in a sorted array and a linear scan for each query term would suffice while comparing to clusters ids in the posting lists.

Remarkably, CS-IIS allows us to update accumulator entries for only those documents that are actually from the best-clusters, thus providing all benefits of IBU strategy discussed in the previous section. Furthermore, there is no need for accessing the cluster-membership information of each document in the posting lists of the query terms as in IBU, as this information is already incorporated into the postings. If the cluster membership information (i.e., document-category

index as discussed in Section 3.3) is kept on disk, this would incur a prohibitive cost for IBU strategy. Our approach with CS-IIS provides a further benefit if the inverted index is compressed, a typical situation in large scale IR systems. In that latter case, CS-IIS would decompress only necessary postings, whereas the IBU strategy would still waste CPU cycles for decoding some postings, just to be discarded when it is realized that they are not from the best-clusters. On the other hand, the CS-IIS slightly increases posting list sizes (due to additional index elements) and thus expected to perform better when the number of documents is much larger than the number of clusters. In the experiments, we show that our claims are justified and the cost of reading longer posting lists is compensated by the in-memory gains during the query processing. Furthermore, gains would be even higher given that many IR systems and Web search engines cache most, even all, of the posting lists (see [105], for example, with a similar assumption).

In the implementation of the skip idea, another alternative is to store the cluster id and skip information at the start of the posting lists. Here we adopt the approach illustrated in Figure 3.2. These two alternatives would have no major difference in terms of posting list I/O time, if the query term posting lists have to be read in their entirety. This is possible, because a term usually appears in enough number of different clusters that would require fetching its whole posting list. However, the former organization can be a viable option if it is possible to evaluate the query by only reading a few cluster blocks from each posting list, in a similar manner to reading only few impact blocks of an impact-sorted list [109]. This alternative organization and its implications especially for the disk access cost and caching performance are left as a future work.

3.5 Experimental Environment

Datasets. In the experiments of this section, we use two datasets. The *Financial Times* collection (1991-1994) of TREC [108] Disk 4, referred to as the FT dataset, and the *AQUAINT* corpus of English News Text, referred to as the AQUAINT dataset, are used in previous TREC conferences and include the actual data, query topics and relevance judgments. During the indexing stage, we eliminated

Table 3.1: Comparison of the characteristics of FT and AQUAINT datasets to some other datasets in the literature (for (*)ed cases, approximate n_c value is calculated using the cover-coefficient-based formula: $n_c = n/x_d$)

Dataset	No. of documents(M)	No. of terms(n)	Avg. no. of distinct terms/doc. (x_d)	No. of clusters (n_c)	Avg. no. of docs./clus. (d_c)
BLISS-1*	152,850	166,216	25.7	6,468	25
MARIAN	42,815	59,536	11.2	5,218	8
INSPEC	12,684	14,573	32.5	475	27
NPL	11,429	7,491	20.0	359	32
FT	210,158	229,748	140.6	1,640	128
AQUAINT	1,033,461	776,820	164.5	5,163	200

English stop-words, and indexed the remaining words, and no stemming is used.

For easy reference, statistical characteristics of the FT and AQUAINT collections are provided in Table 3.1 along with some other databases to give some sense of sizes of the important variables in traditional (INSPEC, NPL), and OPAC (BLISS, MARIAN) [38, 69] collections. In this table the number of clusters, n_c, is obtained by using C^3M clustering algorithm. The numbers show that datasets, more specifically their vector spaces, show various degrees of sparsity as indicated by the number of clusters. For example, FT collection is quite cohesive and the number of clusters is not that high. On the other hand, vector spaces for OPAC (library), BLISS-1 and MARIAN are sparse and contain relatively large number of clusters, since they cover documents in many different subject areas. The content cohesiveness of a dataset may be uniformly distributed and clusters may contain approximately the same number of documents or it can be skewed and it may contain a few number of large clusters containing relatively high number of related documents. We will revisit this issue later in Section 3.6.1.

Queries. We used the TREC-7 query topics (queries 351-400) corresponding to the FT collection along with their relevance judgments. For the AQUAINT dataset, we used the topics and judgments used for TREC 2005 robust track. For the experiments, we created two different types of queries, namely *Qshort* and *Qmedium* that are obtained from these query sets. *Qshort* queries include TREC query titles, and *Qmedium* queries include both titles and descriptions. For the FT query set, we also formed a third query type, *Qlong*, which is created from the top retrieved document of each *Qmedium* query in the query set. A few of the queries do not have any relevant documents in relevance judgment files,

and they are discarded from the query sets. This yielded 49 queries for each of the query sets *Qshort*, *Qmedium* and *Qlong* of FT; and 50 queries for *Qshort* and *Qmedium* sets of AQUAINT. Our query sets cover a wide spectrum from very short Web-style queries (the *Qshort* case) to extremely long ones (the *Qlong* case). Notice that, the latter type of queries can capture the case where a user likes to retrieve similar documents to a particular document and the document itself serves as a query. This provides insight on the behavior of retrieval system at extreme conditions.

Similarity computation. In the following experiments, we used the following term weighting preferences that are reported to yield good retrieval effectiveness in previous works (e.g., [97]). The document term weights are assigned using the *tf-idf* formula whereas query terms are weighted using the augmented normalized frequency formula (see Section 3.2.1). Pre-computed document lengths are employed for normalization. The cosine function is employed for both query-cluster and query-document matching. The selection of centroids and centroid term weigthing are discussed later in this section.

Implementation. The experiments are conducted on a Pentium Core2 Duo 3.0 GHz PC with 2GB memory and 64-bit Linux operating system. All IR strategies are implemented using the C programming language and source codes are available on our Web site[2]. Implementations of the IR strategies are tuned to optimize query processing phase for which we measure the efficiency in the following experiments. In particular, a min-heap is used to select best-clusters and best-documents from the corresponding accumulators as recommended in previous works [118]. Unless stated otherwise, we assume that the posting list per query term is fully brought into the main-memory, processed and then discarded; i.e., more than one term's posting list is not memory resident simultaneously.

[2] `http://www.cs.bilkent.edu.tr/~ismaila/PhD/sources.htm`

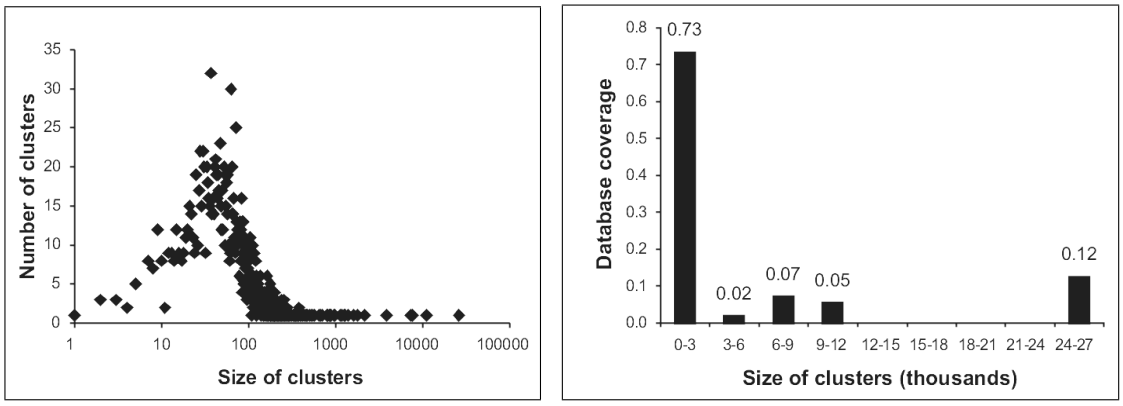
3.6 Experimental Results

In this study, FT and AQUAINT datasets are automatically clustered using the C^3M algorithm. In the following set of experiments, we first investigate the validity of C^3M clustering for the FT collection, and determine some parameters to be used with typical-CBR. Next, we compare the effectiveness of typical-CBR to FS and show that the former is a worthwhile retrieval strategy. In Section 3.6.3.1, we compare the efficiency of typical-CBR strategies described in Section 3.3 for various parameters and identify the most efficient ones. These best-performing strategies and FS (as another baseline) are then compared to our CS-IIS based CBR approach in Section 3.6.3.2. Finally, we give results that reveal the scalability of our findings in Section 3.6.4. Please note that, in the following sections, CBR is interchangeably used with typical-CBR for brevity.

3.6.1 Clustering Experiments

3.6.1.1 Cluster Generation and Characteristics of the Generated Clustering Structure

In this study, C^3M algorithm is used to obtain a flat and non-overlapping clustering of datasets, FT and AUQAINT. For the FT collection, our experiments yield 1,640 clusters. The generated clustering structure follows the indexing-clustering relationships implied by the CC concept. For example, the indexing-clustering relationships $n_c = (m \times n)/t = m/t_g = n/x_d$, and $d_c = t_g$ are all observed in the experiments (for easy reference the values of these variables are repeated here, m =210,158, n =229,748, t =29,545,234, x_d =140.6, t_g = 128.6 and the values obtained for n_c and d_c after clustering are 1,640 and 128). For example, by substituting the corresponding values (m, n, and t) to the above formula, n_c was implied as 1,634 by the relationships, which shows only a 0.4% percent deviation from the real value obtained by actual clustering. Similarly, the d_c (128) value is almost identical with t_g. As shown in the related previous work [35, 36, 42] for a given D matrix the clustering structure to be generated by C^3M is predictable from the indexing characteristics of a database.



(a) (b)

Figure 3.3: Cluster size distribution information: (a) cluster distributions in terms of the number of clusters per cluster size (logarithmic scale), and (b) ratio of total number of documents observed in various cluster size windows.

The size distribution of the clusters is presented in Figure 3.3. In Figure 3.3(a) the x-axis (in logarithmic scale) shows the cluster size in terms of documents and y-axis shows the number of clusters for the corresponding size. The figure reveals that cluster sizes show variety, there are a few large clusters (largest one containing 26,076 documents) and some small clusters, and there are many clusters close to the average cluster size. Figure 3.3(b) shows that majority of the documents (about 73% of them) are stored in clusters with a size 1 to 3,000. Please note that for only 10% of the queries top ten results include documents from the largest cluster, which means that our results are not significantly biased by the existence of a large cluster.

3.6.1.2 Validation of the Generated Clustering Structure

Before using a clustering structure for IR, we must show that it is significantly different from, or better than, random clustering in terms of reflecting the intrinsic nature of the data. Such a clustering structure is called valid. Two other cluster validity issues, clustering tendency and validity of individual clusters, are beyond the scope of this study [65].

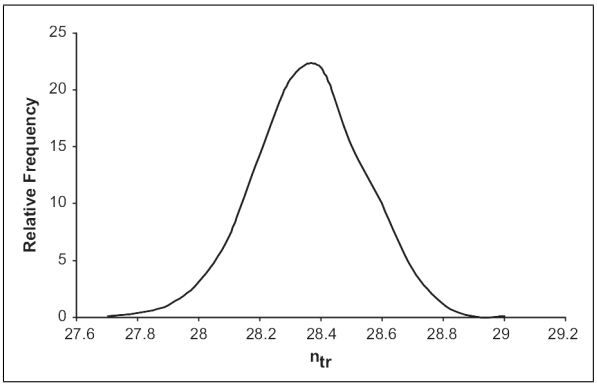


Figure 3.4: Histogram of n_{tr} values for the FT database ($n_t = 20.1$).

Our cluster validation approach is based on the users' judgment on the relevance of documents to queries and follows the methodology defined in [42]. Given a query, a cluster is said to be a *target cluster* if it contains at least one relevant document to the query. Let n_t denote the *average number of target clusters for a set of queries*. Next, let us preserve the clustering structure and distribute all documents randomly to these clusters. The average number of target clusters for this case is shown by n_{tr} and its value can be calculated without creating random clusters by the modified form [42] of Yao's formula [119]; however, we need the distribution of the n_{tr} values for the validity decision. The case $n_t \geq n_{tr}$ suggests that the tested clustering structure is invalid, because it is unsuccessful in placing the documents relevant to the same query into a fewer number of clusters than that of the average random case. The case, $n_t < n_{tr}$, is an indication of the validity of the clustering structure; however, to decide validity one must show that n_t is significantly less than n_{tr}.

According to our validity criterion, we must know the probability density function of n_{tr}. For this purpose, we perform a Monte Carlo experiment and randomly distribute the documents to the cluster structure for 1000 times and for each experiment compute the average number of target clusters. The minimum, maximum, and average n_{tr} values are observed as 27.78, 29.02 and 28.41 (see Figure 3.4 for the probability density function of the n_{tr} values). Then, we compute the n_t value, and it is 20.1. Clearly, n_t is significantly different than the random distributions n_{tr}, since it is less than all of the observed random n_{tr} values. These

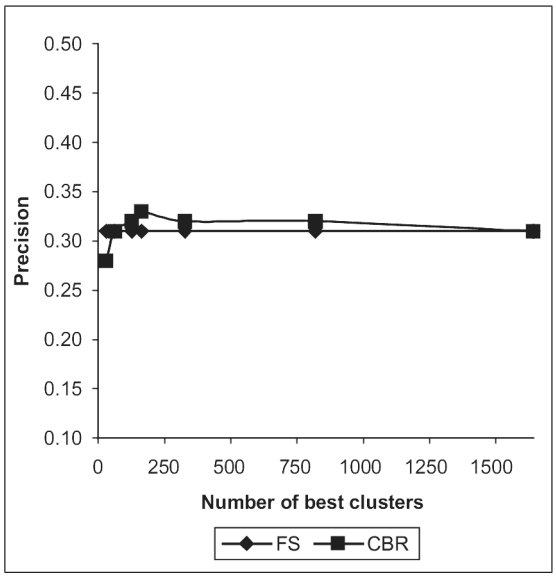


Figure 3.5: MAP versus number of best-clusters (n_s) for $d_s = 10$ and query set $Qmedium$.

observations show that the clustering structure used in the retrieval experiments is not an artifact of the C^3M algorithm, on the contrary, significantly better than random and valid.

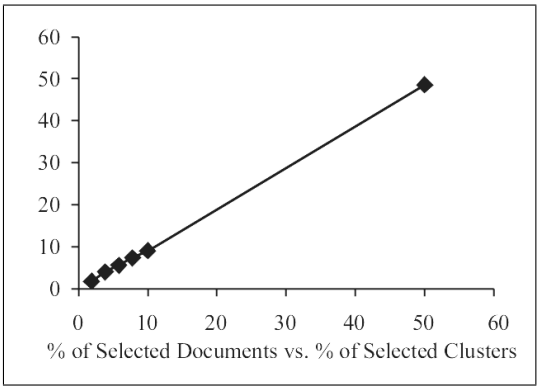
3.6.1.3 Determining the Number of Best-Clusters for CBR

The experiments show that selecting more clusters increases effectiveness since as we increase n_s (i.e., the number of selected clusters) more relevant documents would be covered [95, p. 376]. In a previous work, it was observed that effectiveness increases up to a certain n_s value, after this (saturation) point, the retrieval effectiveness remains the same or improves very slowly [42, Figure 6]. For the INSPEC database, this saturation point is observed when n_s is about 10% of the clusters and during the related experiments about the same percentage of the documents is considered for retrieval. This percentage is typical for (best-match) CBR [95, p. 376].

In our experiments, for a range of n_s values, we retrieved top-10 documents for the query set $Qmedium$ and measured the effectiveness in terms of mean average precision (MAP). The results depicted in Figure 3.5 also confirm the

No. (%) of Selected Clusters	Avg. No. (%) of Selected Documents
32 (1.95)	3857.08 (1.84)
64 (3.90)	8608.14 (4.10)
96 (5.85)	12 041.18 (5.73)
128 (7.81)	15 701.94 (7.47)
164 (10.0)	19 107.12 (9.09)
820 (50.0)	102 016.14 (48.54)
1640 (100.00)	210 158.00 (100.00)

(a)



(b)

Figure 3.6: Relationship between number of selected clusters and number of documents in the selected clusters shown in: (a) table, and (b) plot.

above observation regarding INSPEC, where the effectiveness increases up to 164 clusters (10% of the cluster number n_c for FT dataset) and then no major change occurs. Therefore, we use 10% of n_c as the number of best-clusters in the retrieval experiments.

In Figure 3.6, we report the total number of documents in the clusters for each value of n_s. Figure 3.6(a) and (b) show that, for example, if we select the first best matching 164 clusters (10% of the existing clusters) we need to match 9.09% of the documents with the queries, since this many documents exist in the selected clusters (the numbers are averages for all queries). The observations show that there is a linear relationship between the percentage of clusters selected and the percentage of the documents included in these clusters for FT dataset.

3.6.1.4 Cluster Centroids and Centroid Term Weighting

In the experiments below, we take a simplistic approach and use all cluster member documents' terms as centroid terms for a cluster. One reason for this choice is that, our preliminary experiments with the FT dataset have shown that the effectiveness does not vary significantly for centroid lengths 250, 500 and 1000; whereas using all cluster terms in the centroid yields slightly better performance. Another reason is that by using all cluster terms in centroids, we avoid making

Table 3.2: Term weighting schemes for centroids

Weighting scheme	Term frequency(tf)	Inverse document frequency (idf)
CW1	1	$\ln(\frac{number\ of\ clusters}{number\ of\ centroids\ including\ the\ term}+1)$
CW2	within-cluster term frequency	$\ln(\frac{number\ of\ clusters}{number\ of\ centroids\ including\ the\ term}+1)$
CW3	within-cluster term frequency	$\ln(\frac{sum\ of\ occurrence\ numbers\ in\ the\ clusters}{number\ of\ occurrence\ in\ the\ cluster}+1)$

an arbitrary decision to determine the centroid length. This choice of centroids also enables us being independent of a particular centroid term selection method. Nevertheless, in Section 3.7, we also investigate the performance of different centroid term selection methods for another dataset and show that using all terms of a cluster in its centroid is more effective.

In the rest of this work, we assume that three centroid term weighting schemes are employed: CW1, CW2, and CW3; in all of them the weight of a centroid term is computed by the formula *tf-idf*. In Table 3.2, the three centroid term weighting schemes are summarized. During the best-cluster selection stage of query processing, weights are normalized by using the pre-computed cluster lengths for the corresponding scheme.

3.6.2 Effectiveness Experiments

To evaluate the effectiveness of the IR strategies, namely FS and typical-CBR, top-1000 (i.e., d_s= 1000) documents are retrieved for each of the query sets. The effectiveness results are presented by using the precision at 10 (P@10) and mean average precision (MAP) values (i.e., average of the precision values observed when a relevant document is retrieved) [28] for each of the experiments. For a particular case, we also provide an interpolated 11-point precision-recall graph. All effectiveness figures are computed using the `treceval` software [108].

Table 3.3 provides the P@10 and MAP values for the retrieval strategies. The results essentially reveal that there is no single best approach for IR, and either one of CBR or FS can perform better for different queries. In particular, typical-CBR with CW1 achieves the best performance for short and medium length queries on FT dataset, whereas FS is better for AQUAINT dataset and

Table 3.3: MAP and P@10 values for retrieval strategies ($n_s = 164$ for FT, $n_s = 516$ for AQUAINT, $d_s = 1000$)

Datasets	Query Type	Evaluation metrics	FS	Typical CBR		
				CW1	CW2	CW3
FT	Qshort	MAP	0.107	**0.126**	0.109	0.102
		P@10	0.149	**0.180**	0.149	0.137
	Qmedium	MAP	0.122	**0.134**	0.121	0.113
		P@10	0.163	**0.182**	0.157	0.149
	Qlong	MAP	**0.124**	0.113	0.114	0.109
		P@10	**0.186**	0.176	0.178	0.169
AQUAINT	Qshort	MAP	**0.091**	0.046	0.081	0.071
		P@10	**0.244**	0.176	0.240	0.234
	Qmedium	MAP	**0.100**	0.048	0.089	0.074
		P@10	0.244	0.204	**0.260**	0.248

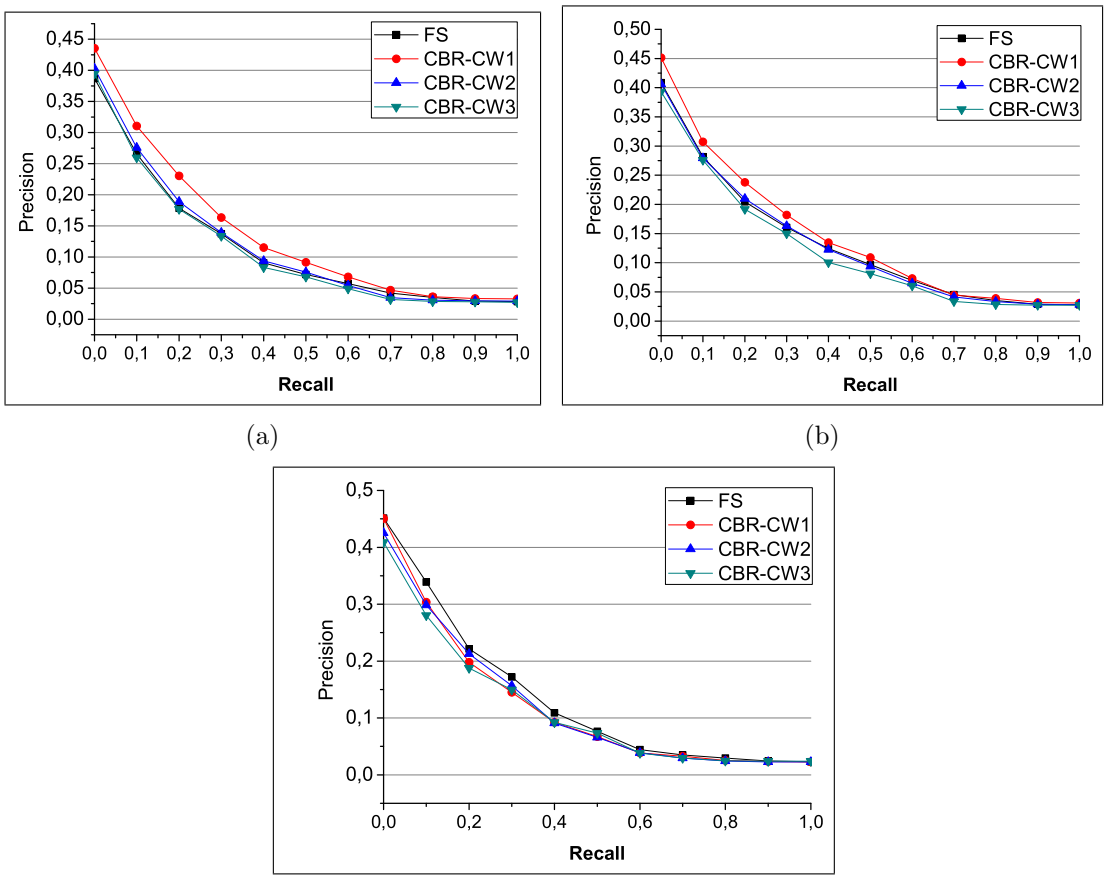
long queries on FT dataset. For a more detailed comparison, consider the 11-point interpolated precision-recall graph given in Figures 3.7 and 3.8 for FT and AQUAINT datasets, respectively. These graphs also imply that the effectiveness of FS and CBR are quite close to each other for different sets of queries with varying lengths. Thus, we conclude that CBR is a valuable retrieval strategy such as FS and improving its efficiency is an important contribution.

Note that, our CBR approaches that blend inverted indexes with cluster based retrieval lead to new opportunities for combining the best results of both strategies, in a way that has not been done before. For example, during query processing we can handle query terms as in FS or CBR like a mixture depending on the query term properties.

3.6.3 Efficiency Experiments

3.6.3.1 Efficiency of Typical-CBR Strategies

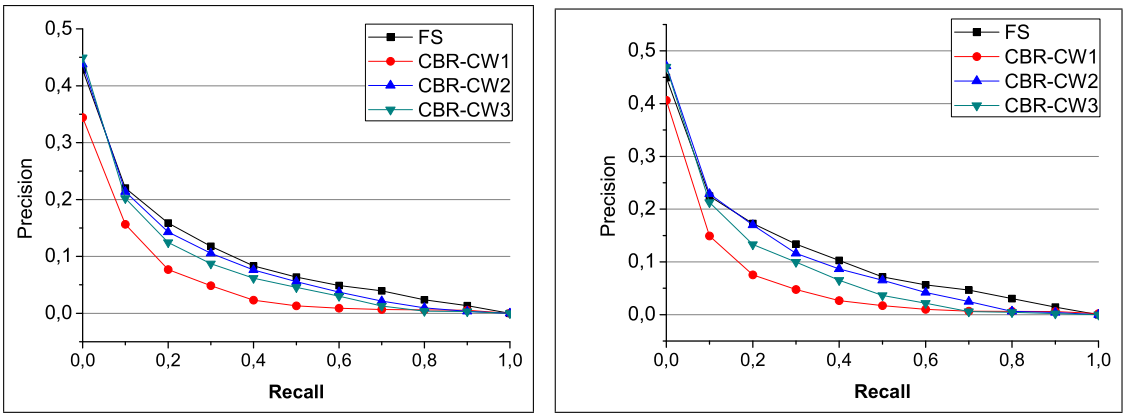
Along with the lines of Section 3.3, we discuss three query processing implementations (IBU, IBI, IAE) and two versions for each such implementation —the version that uses a sorted array (SA) to keep and look up best-clusters, and the version that uses a 0/1 mark array (MA) for the same purpose. During query evaluation, first the queries are matched with the cluster centroids to obtain the best-clusters



(a) (b)

(c)

Figure 3.7: Interpolated 11-point precision-recall graph for IR strategies using FT dataset and (a) *Qshort*, (b) *Qmedium*, and (c) *Qlong*.



(a) (b)

Figure 3.8: Interpolated 11-point precision-recall graph for IR strategies using AQUAINT dataset and (a) *Qshort*, and (b) *Qmedium*.

Table 3.4: Efficiency comparison of the typical-CBR strategies (*IBU*: Intersect Before Update, *IBI*: Intersect Before Insert, *IAE*: Intersect After Extract, *SA*: Sorted Array, *MA*: Mark Array) for FT dataset using CW1

	Time (ms) and operation counts (all averages)	IBU-SA	IBU-MA	IBI-SA	IBI-MA	IAE-SA/MA $L = d_s$	IAE-SA/MA $L = N$
Qshort	Query evaluation time	3	**2**	4	3	3	7
	No. of accumulator updates	908	908	9,792	9,792	9,792	9,792
	No. of nonzero accumulators	848	848	9,462	9,462	9,462	9,462
	No. of intersections	9,792	9,792	9,462	9,462	877	6,931
	No. of heap insertion calls	848	848	848	848	9,462	9,462
Qmedium	Query evaluation time	12	**5**	11	6	7	31
	No. of accumulator updates	3,786	3,786	49,416	49,416	49,416	49,416
	No. of nonzero accumulators	2,899	2,899	39,496	39,496	39,496	39,496
	No. of intersections	49,416	49,416	39,496	39,496	1,000	10,128
	No. of heap insertion calls	2,899	2,899	2,899	2,899	39,496	39,496
Qlong	Query evaluation time	329	89	108	**84**	89	224
	No. of accumulator updates	124,115	124,115	1.8 M	1.8 M	1.8 M	1.8 M
	No. of nonzero accumulators	11,718	11,718	189,510	189,510	189,510	189,510
	No. of intersections	1.8 M	1.8 M	189,510	189,510	1,000	9,503
	No. of heap insertion calls	11,718	11,718	11,718	11,718	189,510	189,510

(top 10% of clusters) as described above. Next, best-documents within these best-clusters and final query outputs are computed using the three possible algorithms with two different data structures (SA, MA) for best-clusters. We measure the efficiency of this latter stage; i.e., selecting the best-documents while filtering out those that are not from the best-clusters. In Tables 3.4 and 3.5, we provide the results for FT dataset for two of the centroid term weighting schemes, namely CW1 and CW2, respectively. The efficiency figures for CW3 are quite similar to CW1 in all cases, and thus not reported here to keep the discussion simple. In Tables 3.6 and 3.7, we give results for AQUAINT dataset, again for CW1 and CW2.

In Tables 3.4 to 3.7, we report in-memory processing time for each strategy, as well as the average number of accumulator update operations, number of nonzero document accumulator entries, number of cluster-id intersection operations and finally number of heap insertion operations, for FT and AQUAINT datasets. Note that, for IAE strategy, we experiment with two different values of the min-heap size (L), namely for $L = N$ (total number of documents) and L = d_s; i.e., 1000. As discussed in Section 3.3, for the latter case, it is possible that less than d_s documents have been retrieved from the best-clusters, which would require rebuilding a larger heap. Still, for Web search scenarios where the user interested

Table 3.5: Efficiency comparison of the typical-CBR strategies (*IBU*: Intersect Before Update, *IBI*: Intersect Before Insert, *IAE*: Intersect After Extract, *SA*: Sorted Array, *MA*: Mark Array) for FT dataset using CW2

	Time (ms) and operation counts (all averages)	IBU-SA	IBU-MA	IBI-SA	IBI-MA	IAE-SA/MA $L = d_s$	IAE-SA/MA $L = N$
Qshort	Query evaluation time	4	**2**	4	3	3	7
	No. of accumulator updates	2,509	2,509	9,792	9,792	9,792	9,792
	No. of nonzero accumulators	2,367	2,367	9,462	9,462	9,462	9,462
	No. of intersections	9,792	9,792	9,462	9,462	877	2,390
	No. of heap insertion calls	2,367	2,367	2,367	2,367	9,462	9,462
Qmedium	Query evaluation time	13	**6**	11	7	7	30
	No. of accumulator updates	13,612	13,612	49,416	49,416	49,416	49,416
	No. of nonzero accumulators	10,200	10,200	39,416	39,416	39,496	39,496
	No. of intersections	49,416	49,416	39,416	39,416	1,000	3,021
	No. of heap insertion calls	10,200	10,200	10,200	10,200	39,496	39,496
Qlong	Query evaluation time	363	145	110	**86**	89	227
	No. of accumulator updates	752,920	752,920	1.8 M	1.8 M	1.8 M	1.8 M
	No. of nonzero accumulators	71,700	71,700	189,510	189,510	189,510	189,510
	No. of intersections	1.8 M	1.8 M	189,510	189,510	1,000	1,595
	No. of heap insertion calls	71,700	71,700	71,700	71,700	189,510	189,510

Table 3.6: Efficiency comparison of the typical-CBR strategies (*IBU*: Intersect Before Update, *IBI*: Intersect Before Insert, *IAE*: Intersect After Extract, *SA*: Sorted Array, *MA*: Mark Array) for AQUAINT dataset using CW1

	Time (ms) and operation counts (all averages)	IBU-SA	IBU-MA	IBI-SA	IBI-MA	IAE-SA/MA $L = d_s$	IAE-SA/MA $L = N$
Qshort	Query evaluation time	23	**10**	26	15	15	71
	No. of accumulator updates	5,289	5,289	81,412	81,412	81,412	81,412
	No. of nonzero accumulators	4,907	4,907	76,596	76,596	76,596	76,596
	No. of intersections	81,412	81,412	76,596	76,596	1,000	13,978
	No. of heap insertion calls	4,907	4,907	4,907	4,907	76,596	76,596
Qmedium	Query evaluation time	92	**28**	84	38	43	299
	No. of accumulator updates	27,237	27,237	401,370	401,370	401,370	401,370
	No. of nonzero accumulators	19,558	19,558	297,885	297,885	297,885	297,885
	No. of intersections	401,370	401,370	297,885	297,885	1,000	11,124
	No. of heap insertion calls	19,558	19,558	19,558	19,558	297,885	297,885

in top-k results where k is usually less than 30 [100], this strategy is very tempting as a reasonably large heap, say of size 1000, would be adequate for most of the cases.

From our findings, the following observations can be drawn.

- First of all, for all strategies, the versions that employ a 0/1 mark array to store best-clusters are faster than their sorted array based counterparts. Of course, the former takes more memory space than the sorted array. However, a hash-table can also be used instead of a mark array, with less space

Table 3.7: Efficiency comparison of the typical-CBR strategies (*IBU*: Intersect Before Update, *IBI*: Intersect Before Insert, *IAE*: Intersect After Extract, *SA*: Sorted Array, *MA*: Mark Array) for AQUAINT dataset using CW2

	Time (ms) and operation counts (all averages)	IBU-SA	IBU-MA	IBI-SA	IBI-MA	IAE-SA/MA $L = d_s$	IAE-SA/MA $L = N$
Qshort	Query evaluation time	26	**13**	28	16	16	68
	No. of accumulator updates	21,960	21,960	81,412	81,412	81,412	81,412
	No. of nonzero accumulators	20,088	20,088	76,596	76,596	76,596	76,596
	No. of intersections	81,412	81,412	76,596	76,596	1,000	1,787
	No. of heap insertion calls	20,088	20,088	20,088	20,088	76,596	76,596
Qmedium	Query evaluation time	99	**39**	86	41	43	294
	No. of accumulator updates	103,883	103,883	401,370	401,370	401,370	401,370
	No. of nonzero accumulators	71,967	71,967	297,885	297,885	297,885	297,885
	No. of intersections	401,370	401,370	297,885	297,885	1,000	1,816
	No. of heap insertion calls	71,967	71,967	71,967	71,967	297,885	297,885

usage and similar efficiency figures (as long as the number of best-clusters is relatively small, which is possible in a practical setting). Thus, in the upcoming sections, we use the versions with MA, unless stated otherwise. Note that, the CBR approach with CS-IIS can employ a sorted array as efficient as a mark array, since the query processing algorithm described in Section 3.4 works in the merge-join fashion while comparing clusters in the postings to the best-cluster set.

- For all query sets IAE approach with a min-heap of size N is inferior to its counterpart with a smaller heap; due to very high costs of building and extracting from the large min-heap. For instance, in Table 3.4, IAE for *Qmedium* takes 31 and 7 milliseconds for a min-heap of size N and d_s, respectively. Everything else being the same, the former approach inserts results to a very large heap, as N = 210,158 here; in contrast to a heap of size d_s, 1000. Also, the former approach extracts 10,128 results from the heap until it obtains top-1000 results from the best-clusters. In contrast, IAE with a heap size of 1000 considers only top-1000 results. For this case, we observed that there is a very slight reduction in P@10 and MAP, so for IAE approach, it is an efficient and effective approach to keep a min-heap of size d_s (or, maybe a slightly larger heap). Note that, since IAE-SA and IAE-MA approaches do not differ significantly in terms of performance, their efficiency figures are shown in the same columns in the tables.

- Assuming that document-cluster index (DC-IIS) is kept in the main memory, the performance of IBU-SA and IBI-SA approaches seem to be very similar, the same is true for the IBU-MA and IBI-MA approaches. IBU-MA approach performs better than IAE-MA and IBI-MA and provides considerable reductions in query processing times for short and medium length queries in all cases. This means that, it is better to use the best-cluster information as early as possible during the query processing, instead of postponing the result integration as in IAE. This allows updating much smaller number of accumulators (i.e., smaller number of query-document similarity computations) and smaller number of insertions to the min-heap; all of which yield a smaller execution time. For instance, in Table 3.6, for *Qshort* set of AQUAINT, IAE-MA and IBI-MA takes 15 ms whereas IBU-MA takes 10 ms, a relative improvement of 33%. For *Qmedium*, there are still important gains; IBU reduces 43 ms of IAE and 38 ms of IBI to only 28 ms. For very long queries (as in the case of *Qlong* set of FT dataset), again IBU and IBI approaches with MA seem to be the most reasonable implementation candidates. In this case, IBU-SA suffers from the excessive cost of cluster-id intersection operations and performs even worse than IAE; so if IBU is the choice of implementation, it should be coupled with MA data structure. Nevertheless, our findings reveal that IBI-MA approach outperforms IBU-MA and seems to be the most efficient approach for long queries.

- If it is impossible to keep DC-IIS in memory, the IAE method with the minimum number of cluster-id intersection operations would be the method of choice. In this case, IBU, the best strategy for short and medium length queries, would be extremely costly. However, we envision that this case may not be highly probable given the modern systems' memory capacities. For instance, in our experimental setup, the size of DC-IIS is only a few MBs.

- Finally, we see that absolute query processing times for cases with CW2 are higher than their counterparts when CW1 is employed. For both dataset and all query sets, CW2 selects larger clusters, since the number of non-zero accumulators considerably increases. The trends discussed above are still

valid, but the relative differences between strategies are less emphasized.

To sum up, we conclude that if document-cluster index (DC-IIS) can be stored in the main memory, it is better to use the best-cluster set information as early as possible during the query processing, and thus IBU, which checks cluster membership before updating the accumulators, performs best for short and medium length query types. If all of the DC-IIS cannot be stored in memory, then IAE (with a reasonable min-heap size) is the best choice. Note that, the latter means that we simply process the query (as if FS) and then filter out the documents at the very end; i.e., do not use best-clusters information during the best-documents selection stage. Since this approach can be implemented on top of an existing IR system without any modification in the query processing algorithm, it is employed as the baseline approach in [30, 31]. In what follows, we use both IBU and IAE (with MA structure) as the baselines to compare to CBR with CS-IIS.

3.6.3.2 Efficiency of Typical-CBR with CS-IIS

In this section, we evaluate the efficiency of typical-CBR with CS-IIS and compare it to CBR strategies IBU and IAE. We also compare it to FS, as a further baseline. In Tables 3.8 and 3.9 we provide in-memory query processing times and number of various operations for the IR strategies for FT and AQUAINT datasets, respectively.

From Tables 3.8 and 3.9, we see that IAE strategy is actually equivalent to FS in the number of in-memory operations; of course, given that best-clusters are already computed or given by the user (as in category-restricted searches of Web directories [30, 31]). IAE has an additional cost of the final result integration (denoted as "number of intersections" in the tables), which is almost negligible. Thus, these two strategies take almost the same amount of time.

On the other hand, CBR with CS-IIS makes the same amount of in-memory operations with IBU strategy, but the former does very few intersections between cluster-ids due to skipping, whereas IBU, as discussed before, should check the cluster of every document in the posting lists of the query terms. In terms of

Table 3.8: Efficiency comparison of typical-CBR with CS-IIS to best performing CBR strategies for FT dataset

		FS	Typical CBR					
			CW1			CW2		
	Time (ms) and operation counts (all averages)		IAE-MA $L = d_s$	IBU-MA	CS-ISS	IAE-MA $L = d_s$	IBU-MA	CS-ISS
Qshort	Query evaluation time	3	3	2	**1**	3	2	**2**
	# of accumulator updates	9,792	9,792	908	908	9,792	2,509	2,509
	# of nonzero accumulators	9,462	9,462	848	848	9,462	2,367	2,367
	# of intersections	0	877	9,792	1,430	877	9,792	1,429
	# of heap insertion calls	9,462	9,462	848	848	9,462	2,367	2,367
Qmedium	Query evaluation time	7	7	5	**2**	7	6	**4**
	# of accumulator updates	49,416	49,416	3,786	3,786	49,416	13,612	13,612
	# of nonzero accumulators	39,496	39,496	2,899	2,899	39,496	10,200	10,200
	# of intersections	0	1,000	49,416	6,652	1,000	49,416	6,651
	# of heap insertion calls	39,496	39,496	2,899	2,899	39,496	10,200	10,200
Qlong	Query evaluation time	87	89	89	**14**	89	145	**53**
	# of accumulator updates	1.8 M	1.8 M	124,115	124,115	1.8 M	752,920	752,920
	# of nonzero accumulators	189,510	189,510	11,718	11,718	189,510	71,700	71,700
	# of intersections	0	1,000	1.8 M	177,220	1,000	1.8 M.	177,670
	# of heap insertion calls	189,510	189,510	11,718	11,718	189,510	71,700	71,700

Table 3.9: Efficiency comparison of typical-CBR with CS-IIS to best performing CBR strategies for AQUAINT dataset

		FS	Typical CBR					
			CW1			CW2		
	Time (ms) and operation counts (all averages)		IAE-MA $L = d_s$	IBU-MA	CS-ISS	IAE-MA $L = d_s$	IBU-MA	CS-ISS
Qshort	Query evaluation time	15	15	10	**6**	16	13	**10**
	# of accumulator updates	81,412	81,412	5,289	5,289	81,412	21,960	21,960
	# of nonzero accumulators	76,596	76,596	4,907	4,907	76,596	20,088	20,088
	# of intersections	0	1,000	81,412	6,093	1,000	81,412	6,092
	# of heap insertion calls	76,596	76,596	4,907	4,907	76,596	20,088	20,088
Qmedium	Query evaluation time	42	43	28	**11**	43	39	**21**
	# of accumulator updates	401,370	401,370	27,237	27,237	401,370	103,883	103,883
	# of nonzero accumulators	297,885	297,885	19,558	19,558	297,885	71,967	71,967
	# of intersections	0	1,000	401,370	26,167	1,000	401,370	26,122
	# of heap insertion calls	297,885	297,885	19,558	19,558	297,885	71,967	71,967

execution times, this provides a great advantage for CS-IIS based strategy; the gains are stable with increasing query length and dataset size. For instance, for using FT dataset and CW1 scheme, it takes 5 and 2 ms to process a medium length query (a relative improvement of 60%) for CBR with CS-IIS and IBU, respectively. In the same case for AQUAINT, the execution times are 28 and 11 ms, still yielding a relative improvement of 60%. Also note that, since cluster membership information is already stored in the posting lists of CS-IIS, there is no need for a separate DC-IIS to be stored in the memory. As before, trends are similar for CW2, but gains are relatively smaller, around 30–40%. Obviously,

Table 3.10: Disk access figures for FT dataset

		IIS	CS-ISS
Qshort	Total list length	9,792	11,229
	Simulated disk access time (ms)	22.6	22.9
Qmedium	Total list length	49,416	56,104
	Simulated disk access time (ms)	81.1	82.5
Qlong	Total list length	1,813,734	1,991,781
	Simulated disk access time (ms)	2,032.4	2,071.4

Table 3.11: Disk access figures for AQUAINT dataset

		IIS	CS-ISS
Qshort	Total list length	81,412	87,520
	Simulated disk access time (ms)	39.7	41.0
Qmedium	Total list length	401,370	427,594
	Simulated disk access time (ms)	168.9	174.6

CBR with CS-IIS outperforms IAE strategy for typical-CBR and also FS (given that best-clusters are pre-computed) with a much wider margin than IBU. For instance, again for AQUAINT and *Qmedium* set, the improvement of CS-IIS approach over FS is around 74% and 51%, for CW1 and CW2, respectively.

Typical-CBR with CS-IIS makes use of the skip elements to determine the cluster of succeeding documents, and thus avoids excessive number of cluster id intersections that occurs in IBU strategy. In return, CS-IIS has longer posting lists than a traditional inverted index. In Tables 3.10 and 3.11, we report the average length of the posting lists that are fetched from the disk and average simulated disk access time. Clearly, FS, CBR strategies IAE and IBU all use the same ordinary document-level index and thus shown in a single column. We prefer to report simulated disk times since it is hard and rather unreliable to make actual measurements due to operating system buffering effects. Simulation parameters are obtained for a Seagate Cheetah ST37405LC disk for which average seek time and latency add up to 8.6 ms and transfer time per sector of 512 bytes is 0.014 ms [63]. Note that, these are compatible with more recent parameters (for instance, [93] uses 4KB blocks with a transfer time of 1 ms, which would yield almost the same results with our parameters).

The results reveal that the average length of CS-IIS lists that are fetched from the disk is only slightly larger than the length of postings of an ordinary index; and the overhead does not exceed 10% in any of the experiments. Therefore, the

increase in the disk access time is well-compensated by the in-memory gains of the CS-IIS in our experimental framework. For instance, for AQUAINT dataset and *Qmedium* query set, average execution times for CW1 (CW2) using IBU and CS-IIS are 28 (39) and 11 (21) ms, respectively (from Table 3.9). So, even when disk access time difference (174.6 - 168.9 $\approx$ 6 ms, from Table 3.11) is added, CS-IIS is still more efficient than its most efficient competitor, IBU. We presume that this would be the case for other datasets, as long as the number of documents is much larger than the number of clusters; which seems like a reasonable assumption.

To sum up, we show that typical-CBR with CS-IIS is considerably more efficient than typical-CBR strategies IBU and IAE. Furthermore, if there is no best-cluster computation cost, e.g., best-clusters are provided by the user; then typical-CBR with CS-IIS can be even more efficient than FS. Note that, other optimizations that are employed for improving the performance of FS in the literature (as discussed in Section 3.2.1.2) can be further adapted to our CS-IIS based approach. In the next chapter, we discuss a modified version of CS-IIS which can compete with FS even when best-cluster computation cost is also involved.

3.6.4 Scalability Experiments

The scalability of C^3M, especially from an incremental clustering point of view, has been thoroughly studied in previous works [35, 38]. In this section, we consider the scalability of our typical-CBR strategy with CS-IIS in terms of its efficiency, effectiveness, and storage structures.

For the scalability experiments we obtained two smaller versions of the FT dataset containing approximately one third and two thirds of the original collection. We refer to them as FT small (FTs) and FT medium (FTm). The characteristics of all FT datasets are given in Table 3.12 (for easy reference the original FT is also repeated in the same table). FTs and FTm, respectively, contain the first 69,507 and 138,669 documents of the original collection. It may be noted in passing that the indexing-clustering relationships are again observed. For example, the indexing-clustering relationship $n_c = n/x_d$ implies 989 and 1345 clusters for the FTs and FTm databases, respectively. The difference between

Table 3.12: Characteristics of the FT Datasets

Dataset	No. of documents(M)	No. of terms(n)	Avg. no. of distinct terms/doc. (x_d)	No. of clusters (n_c)	Avg. no. of docs./clus. (d_c)
FTs	69,507	144,080	145.7	955	73
FTm	138,669	191,112	142.1	1,319	105
FT	210,158	229,748	140.6	1,640	128

Table 3.13: MAP and P@10 values for retrieval strategies using the subsets of FT dataset for *Qmedium* ($n_s = 10\%$ of n_c, $d_s = 1000$)

Dataset	Evaluation metrics	FS	Typical CBR		
			CW1	CW2	CW3
FTs	MAP	**0.053**	0.053	0.053	0.046
	P@10	**0.131**	0.127	0.131	0.118
FTm	MAP	0.088	**0.094**	0.090	0.088
	P@10	0.139	**0.159**	0.139	0.145
FT	MAP	0.122	**0.134**	0.121	0.113
	P@10	0.163	**0.182**	0.157	0.149

actual numbers and projected numbers is less than 4% as in the case of FT.

In the scalability experiments, as a representative case, we only consider the *Qmedium* query set, which is the mid-way in terms of the query sizes we used. In the experiments we again retrieve 10% of the clusters ($n_s = 0.1 \times n_c$), examine the top-1000 documents (d_s= 1000) for performance measurement, and use centroids with all terms of clusters as in the previous experiments.

3.6.4.1 Scalability of Effectiveness

The experimental results in terms of P@10 and MAP are reported here. Table 3.13 shows that when we use the small database, FTs, the CBR effectiveness is comparable to that of FS. In the case of FTm and FT the performance of CBR improves and outperforms FS. These observations confirm that our CBR methodology scales well with the collection size. This improvement of CBR effectiveness can be attributed to the refinement of cluster structures with increasing collection size for FT.

Table 3.14: Efficiency comparison of typical-CBR with CS-IIS to best performing CBR strategies for FTs dataset and *Qmedium*

	FS	Typical CBR					
		CW1			CW2		
		IAE-MA $L = d_s$	IBU-MA	CS-ISS	IAE-MA $L = d_s$	IBU-MA	CS-ISS
Query evaluation time	2	3	1	**1**	3	2	**1**
No. of accumulator updates	16,876	16,876	1,479	1,479	16,876	6,023	6,023
No. of nonzero accumulators	13,441	13,441	1,118	1,118	13,441	4,556	4,556
No. of intersections	0	993	16,876	3,059	993	16,876	3,053
No. of heap insertion calls	13,441	13,441	1,118	1,118	13,441	4,556	4,556

Table 3.15: Efficiency comparison of typical-CBR with CS-IIS to best performing CBR strategies for FTm dataset and *Qmedium*

	FS	Typical CBR					
		CW1			CW2		
		IAE-MA $L = d_s$	IBU-MA	CS-ISS	IAE-MA $L = d_s$	IBU-MA	CS-ISS
Query evaluation time	4	5	3	**2**	5	4	**3**
No. of accumulator updates	32,917	32,917	2,491	2,491	32,917	10,346	10,346
No. of nonzero accumulators	26,295	26,295	1,892	1,892	26,295	7,804	7,804
No. of intersections	0	1,000	32,917	4,848	1,000	32,917	4,846
No. of heap insertion calls	26,295	26,295	1,892	1,892	26,295	7,804	7,804

3.6.4.2 Scalability of Efficiency

Tables 3.14 and 3.15 provide the in-memory processing times and number of operations for various IR strategies that are obtained for FTs and FTm datasets, respectively. The numbers reported for the in-memory operations increase almost linearly with the dataset size; and the CS-IIS based approach is still superior to others. However, the improvement of CS-IIS over other strategies does not behave perfectly linear. This is due to the fact as the datasets get smaller, the absolute time values also become very small and somewhat inaccurate to measure. Nevertheless, we can still claim that the number of operations and their reflection to time is scalable.

For the sake of completeness, in Tables 3.16 and 3.17, we report the average posting list lengths that are fetched from the disk and simulated disk access times. Again, these values scale well. For disk read times, please note that average disk seek and rotational delay time for *Qmedium* set is around 70 ms, and the remaining time values are for sequential reading of posting lists; namely,

Table 3.16: Disk access figures for FTs dataset and *Qmedium*

	IIS	CS-ISS
Total list length	16,876	19,952
Simulated disk access time (ms)	73.4	74.1

Table 3.17: Disk access figures for FTm dataset and *Qmedium*

	IIS	CS-ISS
Total list length	32,917	37,799
Simulated disk access time (ms)	77.4	78.5

3.4, 7.4 and 12.5 ms as obtained for FTs (Table 3.16), FTm (Table 3.17) and FT (Table 3.10). Thus, simulated disk read times also increase linearly with the collection size.

3.6.4.3 Scalability of Storage Space

In Table 3.18, we compare the storage space used of CS-IIS and ordinary document-level IIS for the FT datasets and AQUAINT. As we increase the size of the database, the cost of storing CS-IIS slightly decreases (from 0.30 to 0.26) with respect to IIS. This is due to the fact that the rate of increase in the number of clusters is smaller than that of documents (see Table 3.12). Note that, while these values are for the uncompressed case, most large scale IR systems and Web search engines store inverted files in the compressed form. In Chapter 4, we investigate this issue and show that storage overhead of CS-IIS can be further reduced.

3.6.5 Summary of the Results

In Section 3.6, we provided an evaluation of typical-CBR in an environment where the documents are automatically clustered by using a partitioning algorithm

Table 3.18: Storage requirements (in MB) for inverted index files

Storage Component	FTs	FTm	FT	AQUAINT
IIS	77	150	225	1,360
CS-IIS	101 (+31%)	190 (+27%)	284 (+26%)	1,630 (+20%)

(C^3M). During query processing, best-clusters set for each query is automatically determined as the clusters that yield the highest query-cluster similarity. The size of best-clusters set is set to the 10% of the total number of clusters in the collection. From the experiments, we draw the following conclusions:

- Both retrieval models, namely FS and typical-CBR, achieve similar effectiveness values.
- We discuss three strategies for typical-CBR, and show that exploiting the best-clusters set information as early as possible during the query processing improves in-memory execution performance. Furthermore, our CS-IIS based typical-CBR strategy outperforms all three CBR strategies, and even FS; if there is no best-cluster computation time involved. This is possible for the cases where the user manually determines the best-clusters to be searched.
- The posting lists for CS-IIS are slightly longer than the lists of an ordinary index due to additional information stored; however, the increase in sequential read times are compensated by the in-memory gains, as long as the number of documents is much larger than the number of clusters. This overhead could be further reduced by the OS buffering effects and posting list caching techniques employed in search engines (e.g., see [18]).
- The results are independent of the centroid lengths and weighting schemes, as the variations over these parameters do not significantly affect the presented results.
- Disk storage space for CS-IIS is only moderately higher than a traditional index, and current compression techniques may further reduce this overhead. In CS-IIS, such a reduction has the potential of further improving the processing time, since by using our skipping approach the decompression time can be reduced significantly. We explore these directions in Chapter 4.
- The experiments show that our results are scalable. Effectiveness of CBR slightly increases while the efficiency improvements of CBR with CS-IIS remain stable with increasing collection size.

3.7 Case Study I: Performance of Typical-CBR with CS-IIS on Turkish News Collections

In this section, we present an application of typical-CBR and another set of CBR experiments using the largest Turkish IR test collection in the literature. Our goal is to verify our findings that are discussed in the previous sections and put our ideas to work by building a practical Turkish news portal that allows cluster-based searches. To our knowledge, our work presented here involves the experiments for automatic document clustering and CBR using the largest available corpora in Turkish IR literature. To this end, in the following experiments we investigate the effectiveness and efficiency implications of

- cluster centroid term selection and weighting mechanisms,
- automatic clustering and manual classification of documents,
- alternative strategies for typical-CBR, and
- employing CS-IIS for typical-CBR.

3.7.1 Experimental Setup

Dataset. We use the recently constructed *Milliyet* dataset for Turkish along with the TREC-style query and relevance judgments sets [39, 40]. The dataset includes 408,305 documents. Following the findings in an earlier study [39], we eliminated the stopwords and then stemmed the remaining terms using a simple 5-prefix stemmer. After stemming, the dataset includes 180,000 distinct terms including numbers. The query set includes 72 queries with 14.4 terms on the average. For query-document matching, we use a variant of the cosine function that has shown to yield the best effectiveness results for this dataset [39].

Clusters, centroid term selection and weighting. We again automatically cluster the dataset using C^3M algorithm [37] in partitioning mode, which yields 1,357 clusters. We also use a manual classification of newspaper articles as provided by the publisher Web site (e.g., economics, art, politics, etc.), which includes

Table 3.19: Bpref figures for CBR strategies with different centroid term selection and weighting methods

	AllSel		LogSel		AvgSel	
	MAN	AUT	MAN	AUT	MAN	AUT
CW2	**0.35**	**0.40**	0.33	0.39	0.29	0.39
CW3	0.27	0.40	0.02	0.09	0.01	0.10

12 classes. In the following, these clustering structures are referred to as AUT and MAN, respectively. We investigate several different approaches for determining centroid terms of each cluster (class). We name these selection strategies as follows (see [107] for details):

- *All terms (AllSel):* All terms that are in the clusters are employed as centroid terms. This is the approach employed in Section 3.6.
- *Log selected terms (LogSel):* The terms that appear in a number of documents that is larger than the value $\log_2(C)$ documents (where C is the cluster size) are selected as centroid terms.
- *Average selected terms (AvgSel):* The selected terms are those that have a total frequency in a cluster which is larger than the average of all term frequencies in that cluster.

We only employ CW2 and CW3 centroid term weighting schemes, as described before (see Table 3.2).

3.7.2 Experimental Results

In Table 3.19, we compare the centroid term selection and weighting methods in terms of effectiveness. For automatic (manual) clustering, *AllSel*, *LogSel*, *AvgSel* selection methods yield 4,419 (42,208), 755 (15,036) and 915 (4,174) distinct centroid terms on the average, respectively. In the following experiments, we use *AllSel* method with CW2, which leads to the highest effectiveness for both MAN and AUT cases.

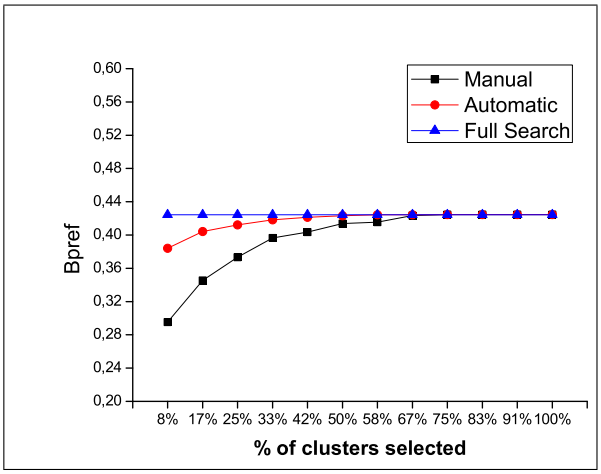


Figure 3.9: Bpref figures of CBR for varying percentages of selected clusters.

In Figure 3.9, we illustrate the CBR effectiveness figures of automatic clustering (AUT) and manual classification (MAN) for varying percentages of selected best-clusters. As MAN includes only 12 clusters, we consider cases where percentage of best-clusters start from 1/12 (8%) and increases by 8%. It is seen that, when best-clusters are 17% of the all clusters, AUT case achieves comparable bpref scores with full-search (i.e., 0.40 vs. 0.42, respectively). MAN case cannot reach to the same bpref figures until almost 33-42% of all clusters are selected. We think that this is due to the skewness of the data distribution in MAN. That is, some of these clusters, such as politics or economics, are very crowded whereas some others like magazine or astrology include relatively few news stories.

Finally, in Table 3.20, we provide in-memory efficiency figures when 17% of the clusters are selected as best-clusters. We exclude best-cluster selection time, as before. The results reveal that, FS, which takes 0.134 seconds, is only slightly more efficient than IAE strategy for CBR (0.136 and 0.139 sec. for MAN and AUT, respectively). Note that, in this case, IBU and IBI are also inferior to IAE (and FS). We think that this may be explained by choosing a larger number of clusters as best-clusters in this framework. Furthermore, there are only 1,357 cluster for 400K documents, whereas FT dataset yielded 1,640 clusters for 200K documents (see Table 3.1). This implies that clusters are larger and imbalanced. Finally, the query lengths for this setup are much longer (i.e., 14.4 terms on average, whereas *Qmedium* set for FT and AQUAINT datasets include 8.2 and

Table 3.20: In-memory query processing efficiency for IR approaches (in ms)

Clustering structure	FS	Typical-CBR			
		IBU	IBI	IAE	CS-IIS
MAN	134	152	157	136	**98**
AUT		191	166	139	**126**

9.4 terms on average, respectively). Nevertheless, Table 3.20 reveals that CBR with CS-IIS, our major contribution in this chapter, still outperforms all of them.

3.8 Case Study II: Performance of Typical-CBR with CS-IIS on Web Directories

In the previous Sections 3.6 and 3.7, we evaluated CBR effectiveness and efficiency on document collections that are automatically clustered by a partitioning algorithm (C^3M), which yields a flat clustering structure. In contrast, Web directories typically involve a hierarchy of categories and employ human editors who assign Web pages to corresponding categories. Web surfers make use of such directories either for merely browsing, or issuing a query under a certain category that they have chosen (i.e., a category-restricted search [30, 31]).

The major contribution of this section is demonstrating how CS-IIS can be employed in a hierarchical clustering framework, such as a Web directory, and how exactly the gains or costs are affected due to some unique properties of this framework. The experiments are held using the largest available Web directory dataset as provided by Open Directory Project (ODP). This work differs from the earlier experiments presented in this chapter in the following ways: i) in Sections 3.6 and 3.7, an automatic and partitioning clustering structure is assumed, whereas the Web directory domain involves a hierarchical taxonomy, ii) the previous sections involve moderate number of categories (although they were quite large figures in the automatic text clustering literature) whereas Web directories involve hundreds of thousands of categories, and iii) both the data, categorization and queries are generated by real users, which makes this environment a unique opportunity to show the applicability of the CS-IIS approach.

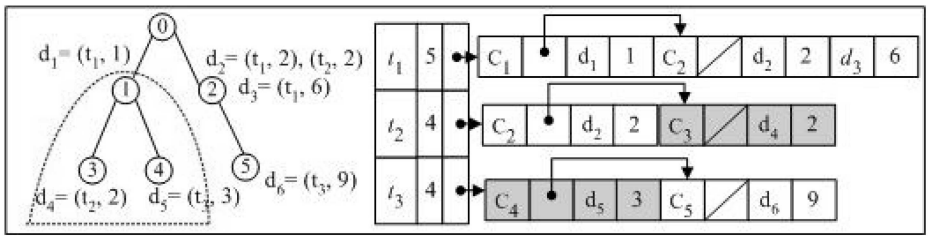


Figure 3.10: A hierarchical taxonomy and the corresponding CS-IIS. Given the query = $\{t_2, t_3\}$ that is restricted to C_1, the query processor first identifies the target categories (C_1, C_3 and C_4, as shown within dotted lines) and then processes posting lists. Note that, only the shaded parts of the posting lists are processed and the rest is skipped.

Note that in the following sub-sections, we use the term "category" instead of "cluster". To be consistent with the terminology of Web directories, we also refer to best-clusters set as the "target categories". In this framework, the user specifies an initial category for his/her query, and the target categories are those under the user specified category; i.e., the sub-tree (or graph, more generally) rooted at the user's initial category selection (e.g., see [30]). We call this a category-restricted search (as in [30, 31]).

While constructing the CS-IIS for a hierarchy as in the case of a Web directory, documents in a posting list are grouped with the categories under which they immediately appear (see Figure 3.10). This is different from an earlier proposal where the signature of the full category path is stored for each document [31]. Once CS-IIS is constructed, query processing proceeds as described in previous sections.

3.8.1 ODP Dataset Characteristics and Experimental Setup

Dataset. For this study, we use the largest publicly available category hierarchy as provided by ODP Web site [85]. After preprocessing and cleaning data files, we end up with a category hierarchy of approximately 719K categories and 4.5 million URLs. For most of the URLs, a one- or two-sentence length description is also provided in the data file. In this work, we use these descriptions as the

actual documents. Note that, this yields significantly shorter documents (with a few words on the average) than usual.

While constructing the hierarchy using the data files, we decided to use `narrow`, `symbolic` and `letterbar` tags in the data file as denoting the children of a category. The resulting hierarchy is more like a graph than a tree in that only 36% of the categories have a single parent. This indicates that, it would be better to keep track of the immediate category of a document as in our CS-IIS (and also the approach in [30]) with respect to keeping the entire path (e.g., see [31]), as there may be several paths to a particular document.

We find that a great majority of categories are rather small; i.e., 98% of them includes less than 50 documents. Furthermore, 93% of the documents (about 4.2 million) belong to only one category, whereas 6% of the documents belong to two parents and only the remaining 1% of the documents appears in three or more categories. These numbers are important for CS-IIS, since a posting list needs to store the same documents as many times as they appear in different categories. The above trends conform the observations in earlier works [30, 31], and show that the waste of storage space due to overlapping documents among categories would not be high.

Indexing. After preprocessing, the document description file takes 2 GB on disk. During inverted index creation, all words (without stemming) are used except numbers and stopwords, yielding 1.1 million terms at the end. The resulting size of the ordinary inverted file (i.e., to be used by the baseline approach) is 342 MB whereas the size of the CS-IIS file is 609 MB. Note that, the additional space used in CS-IIS is unusually large in comparison to our previous findings (i.e., in Section 3.6, only 26% and 10% more space usage was observed for FT and AQUAINT datasets, respectively). We attribute two reasons for this outcome, and state their remedies as follows: i) the dataset includes too many categories with respect to the number of documents. In Section 3.6, for instance, AQUAINT collection of approximately 1M documents yields only 5,163 clusters, whereas here approximately 4.5M pages are distributed to 719K categories. We believe this situation would change for our benefit in time, as the growth rate of hierarchy may

possibly be less than that of the collection. Furthermore, the taxonomy may be populated to reach to a much larger collection size using automatic classification techniques. ii) the documents are unusually short, as we use just the summaries in this initial stage of our work.

Queries and query processing efficiency. We use two methods for obtaining category-restricted queries. First, we prepared a Web-based system which allows users (graduate students) to specify queries along with categories and evaluate the results.

For this experiment, we use 64 category-restricted queries from this system and refer to them as *manual-category* queries. Additionally, we employ the efficiency task topics of TREC 2005 terabyte track. This latter set includes 50K queries, and 46K of them are used in the experiments after those without any matches in the collection are discarded. This set is referred to as *automatic-category* queries.

Notice that, the latter query set lacks any initial target category specification, so we had to match the queries to categories automatically as discussed in the previous sections. To achieve this, we again use all terms in categories as centroids to compute query-category similarities. At this stage, the well-known *tf-idf* term weighting with the cosine measure is employed. Next, for each query, we find the top-10 highest scoring category and choose a single one with the shortest distance to the root (i.e., imitating the typical user behavior of selecting a category as shallow as possible [31] while browsing).

For both query sets, this initial target category is then further expanded; i.e., the sub-graph is obtained. In the following experiments, the time cost for obtaining target categories is not considered, as this stage is exactly the same for both of the compared strategies and can be achieved very efficiently by using the method in [30]. The query-document matching stage also uses the *tf-idf* based weighting scheme and cosine similarity measure as described in Section 3.2.1. Top-100 results are returned for each query.

Table 3.21: In-memory query processing efficiency (all average values, relative improvement in query execution time by CS-IIS is shown in parantheses)

Query set	Time (ms) and operation counts (all averages)	Baseline CBR (IAE)	CBR with CS-IIS
Manual-category	Query evaluation time	128	**109 (15%)**
	No. of nonzero accumulators	17,219	11,758
	No. of postings fetched	17,339	28,913
Automatic-category	Query evaluation time	158	**100 (37%)**
	No. of nonzero accumulators	19,900	250
	No. of postings fetched	20,367	33,271

3.8.2 Experimental Results

The in-memory average query processing (CPU) times are reported in Table 3.21, as well as the number of non-zero accumulators and the average length of posting lists fetched. For these set of experiment, we only employ IAE strategy for CBR as the baseline, following the practice in the literature [30, 31].

Table 3.21 reveals that for both query sets, using CBR with CS-IIS improves the efficiency of work done in main memory. This gain is caused by two factors: first, skipping irrelevant clusters reduces the redundant partial similarity computations. Secondly, but equally importantly, the number of non-zero accumulators at the end of query, which are to be inserted into and extracted from a min-heap, is considerably reduced. We even favor the IAE strategy by assuming that the document-category index (DC-IIS) is in the memory. Note that, the gains would be more emphasized if compression had been used, as skipping would also reduce the burden of decoding operations as we discuss in Chapter 4. A second observation is that, the manual-category queries apparently cover a larger sub-graph and thus process more documents for both strategies. Indeed, in that query set, 55% of the queries are restricted to categories at depth 1. In contrary, the automatic-category queries usually locate the initial target category in a deeper position in the graph. That is why the latter makes much less operations and obtains more gains. Nevertheless, we used the same automatic category computation technique for the manual-category query set, and observed that most of the returned categories are reasonably relevant to queries, but not necessarily the same as the ones as specified by the user. Our current work involves a quantitative analysis of target category selection and using more sophisticated term weighting schemes

to represent categories.

For the disk access issues, we assume that posting lists are brought to memory entirely and discarded once they are used (i.e., no caching). In Table 3.21, the difference between the list lengths fetched from the disk is around 12 K postings (for manual-category query set), adding up to 96 KB (i.e., 8 bytes/posting). Considering a typical disk with the transfer rate of 20 MB/s, the additional sequential read cost is only 5 ms, which is clearly less than the in-memory gains for this case.

3.8.3 Discussions and Summary

For Web directories, typical-CBR with CS-IIS has some other advantages in comparison to the earlier works in the literature. We observe that the real life hierarchies are quite large (in contrast to those in [30, 31] as discussed in Section 3.2.3.5). So, it may be difficult to use the signature-file based system as in [31]. The approach discussed in [30] enforces an upper limit on the number of categories (e.g., 1024). Furthermore, both of these earlier works involve using a part of document id to represent its categories, which would require bitwise operations during query processing and may complicate the use of typical index compression schemes. On the other hand, CS-IIS imposes no limits on neither the size of category nor the number of documents and can be practically used in existing systems, even with compression. This latter issue is further discussed in Chapter 4.

In this section, the CS-IIS is adapted for hierarchical categories in Web directories to allow efficient processing of category-restricted queries. Our current results show that, despite the use of very short document descriptions and the imbalance between the number of categories and documents, the proposed strategy is quite promising.

3.9 Conclusions

In this chapter of the thesis, we investigated the effectiveness and efficiency of cluster-based retrieval for various clustering scenarios and using several parameters. We showed that CBR is a worthwhile retrieval technique as an alternative or complementary approach to FS. To improve the efficiency of typical-CBR, we first proposed some alternative query processing techniques. Next, as the most essential contribution of this chapter, we introduced a cluster-skipping inverted index structure (CS-IIS) that is shown to be superior to the other CBR strategies that use an ordinary document-level index. We presented a wide range of experiments involving automatically clustered and manually categorized datasets, and automatically and manually determined best-cluster sets. In all cases, CS-IIS provides significant improvements for the in-memory (CPU) time efficiency. Furthermore, under the realistic assumption, we showed that the slightly larger disk access cost of CS-IIS can also be compensated by the aforementioned gains.

Finally, we emphasize that typical-CBR with CS-IIS can be even more efficient than FS, if the best-cluster computation time is not involved. In the next chapter, we further improve our CS-IIS data structure so that the above restriction can be relaxed. We will also provide efficiency results in a framework where all index files are stored in a compressed form, a practice which is possibly adapted by all large scale IR systems and Web search engines.

Chapter 4

Search Using Document Groups: Incremental Cluster-Based Retrieval

In this chapter, we propose a modified version of our cluster-skipping inverted index structure (CS-IIS) and a new, incremental, cluster-based retrieval (CBR) approach. In Section 4.1, we discuss the motivation for this part of our research and list our major contributions. Next, in Section 4.2, we introduce the incremental-CBR strategy that operates on top of the CS-IIS, which is enriched to include cluster centroid information. In Section 4.3, we discuss the compression of the CS-IIS with an emphasis on the benefits of document id reassignment in our framework. Sections 4.4 and 4.5 are devoted to experimental setup and results, respectively. We extensively evaluate the proposed strategy and compare to an enhanced FS implementation based on dynamic pruning and skips [79]. In Section 4.6, we show that our new strategy coupled with CS-IIS can be used in a dynamic pruning framework for Web search engines, where the documents are simply clustered according to their Web sites. Finally, we conclude and point to future work directions in Section 4.7.

4.1 Introduction

In Chapter 3, we have introduced the CS-IIS to improve the efficiency of second stage of typical-CBR; i.e., selecting the best-documents that belong to the best-clusters. In this chapter, we attempt to optimize both stages of typical-CBR so that almost no redundant work is done. That is, our goal is to design a CBR strategy that can overcome the efficiency weaknesses of typical-CBR and be as efficient as FS while still providing comparable effectiveness with FS and typical-CBR. We envision that our new CBR approach can either be used in environments that are inherently clustered/categorized due its own application requirements (such as the Web directories or digital libraries), or in the cases where the collection is clustered essentially for the purposes of search efficiency; i.e., as another dynamic pruning technique for FS (see Section 3.2.1.2 for others).

We propose some modifications on CS-IIS and based on this structure introduce a new CBR strategy. In the modified CS-IIS file, in addition to the cluster membership information, within-cluster term frequency information is also embedded into the inverted index. By this extension, centroids are now stored along with the original term posting lists. This enhanced inverted file eliminates the need for accessing separate posting lists for centroid terms (recall that typical-CBR uses a separate centroid IIS for best-cluster selection, as shown in Figure 3.1). In the new CBR method, the computations required for selecting the best-clusters and the computations required for selecting the best-documents of such clusters are performed together in an incremental and interleaved fashion. The query terms are processed in a discrete manner in non-increasing term weight order. That is, we envision a term-at-a-time query processing mode in this work; whereas another highly efficient alternative, document-at-a-time is out of scope [15]. As we switch from the current query term to the next, the set of best-clusters is re-computed and can dynamically change. In the document matching stage of CBR only the portions of the current query term posting list corresponding to the latest best-clusters set are considered. The rest is skipped, hence is not involved in document matching. During document ranking, only the members of the most recent best-clusters set with a non-zero similarity to the

query are considered.

In the literature, it is observed that the size of an inverted index file can be very large [118]. As a remedy, several efficient compression techniques are proposed that significantly reduce the file size. In this chapter, we concentrate on the IR strategies with compression where the performance gains of our approach become more emphasized. Indeed, our incremental-CBR strategy with the new inverted file is tailored to be most beneficial in such a compressed environment. That is, skipping irrelevant portions of the posting lists during query processing eliminates the substantial decompression overhead (as in [79]) and provides further improvement in efficiency. In compression, we exploit the use of multiple posting list compression parameters and reassign document ids of individual cluster members to increase the compression rate, as recently proposed in the literature [23, 102].

The proposed approach promises significant efficiency improvements: If the memory is scarce (say, for digital libraries and proprietary organizations) and the index files have to be kept on disk, the incremental-CBR algorithm with modified CS-IIS allows the queries to be processed by only one direct disk access per query term (assuming that a posting list is read entirely at once). Furthermore, even if the centroid and/or document index is stored in memory, which is probable with the recent advances in hardware (see [105], as an example), the CS-IIS saves decoding and processing the document postings that are not from best-clusters, a non-trivial cost. We show that, the most important overhead of CS-IIS, longer posting lists, is reduced to an affordable overhead by our compression heuristics and even with a moderate disk, the gains in efficiency can compensate for the slightly longer disk transfer times (given that the number of clusters tends to be much smaller than the number of documents, as discussed in the previous section).

Our comparative efficiency experiments cover various query lengths and both storage size and execution time issues in a compressed environment. The results even with lengthy queries demonstrate the robustness of our approach. We show that our approach scales well with the collection size. In the experiments,

we use multiple query sets and three datasets of sizes 564MB, 3GB and 27GB, corresponding to 210,158, 1,033,461 and 4,293,638 documents, respectively.

4.1.1 Contributions

Our contributions in this chapter are:

- *Introducing a pioneering CBR strategy:* we introduce an original CBR method using an enriched cluster-skipping inverted index structure and refer to it as incremental-CBR. The proposed strategy interleaves query-cluster and query-document matching stages of typical-CBR for the first time in the literature.

- *Embedding the centroid information in document inverted indexes:* For memory-scarce environments (e.g., private networks, digital libraries, etc.) where the index files should be kept on disk, we eliminate disk accesses needed for centroid inverted index posting lists by embedding the centroid information in document posting lists. This embedded information enables best-cluster selection by only accessing the document inverted index. By this way during query processing, each query term requires only one direct disk access rather than separate disk accesses for centroid and document posting lists. (We assume that a posting list for a term is entirely fetched once it is located on the disk. It is also possible to read a posting list in a block-by-block manner for some index organizations and early pruning purposes. Even in this case, embedding centroid information may allow one less direct disk access. In such a setup, alternative organizations of CS-IIS can also be possible, e.g., by sorting the cluster blocks in each list according to some importance score and then reading the list in a blockwise manner. These directions are left as a future work.)

- *Outperforming full search (FS) efficiency:* we show that for large datasets incremental-CBR outperforms FS (and the IAE strategy, which usually

serves as a baseline typical-CBR strategy as discussed in the previous chapter) in efficiency while yielding comparable (or sometimes better) effectiveness figures. We also show that efficiency of our approach scale well with the collection size. The proposed approach is also superior to an enhanced implementation of FS approach that employs the "continue" pruning strategy accompanied with a skipping IIS, as described in [79].

- *Adapting the compression concepts to a CBR environment*: we adapt multiple posting list compression parameters and specify a cluster-based document id reassignment technique that best fits the features of CS-IIS.

- *CBR experiments using a realistic corpus size with no user behavior assumption:* we use the largest corpora reported in the CBR literature, assume no user interaction, and perform all decisions in an automatic manner. Only a few studies on CBR use collections as large as ours (e.g., [73]).

4.2 Incremental-CBR with CS-IIS

4.2.1 CS-IIS with Embedded Centroids

A cluster-skipping inverted index structure (CS-IIS) differs from a typical IIS since in posting lists it stores the documents of each cluster in a group adjacent to each other. It contains a skip-element preceding each such group to store the id of cluster to which the following document group belongs, and a pointer to the address where the next skip-element can be accessed in the posting list. In Chapter 3, it is shown that cluster-skipping in query processing improves the query processing time. Furthermore, since cluster membership information is embedded into the IIS, it needs no separate cluster membership test as it is required in other typical-CBR methods (such as those in Section 3.3).

In this work, we introduce an enriched cluster-skipping IIS, which contains an additional centroid-element for each cluster in a given posting list (Figure 4.1). Note that, in Figure 4.1, the same D matrix and clusters of Figure 3.2 are used to emphasize similarities and differences between the two skip approaches. The

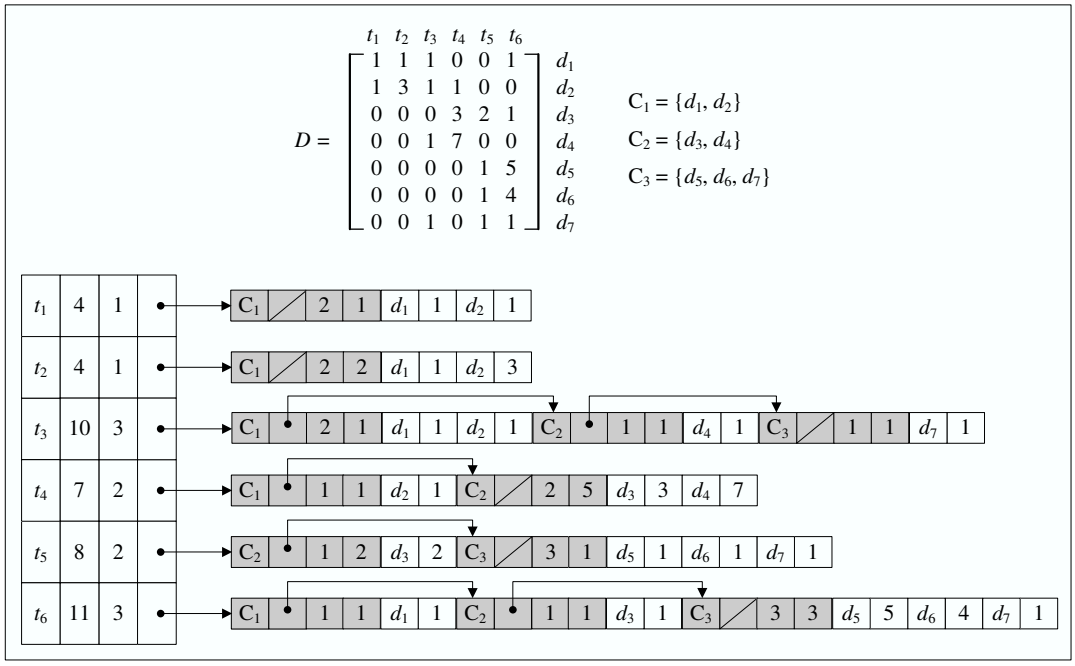


Figure 4.1: Cluster-Skipping Inverted Index Structure (CS-IIS) (embedded skip- and centroid-elements are shown as shaded).

new centroid-element stores: i) the number of documents (i.e., sub-posting list length, explained later), and ii) average within-cluster term frequency ($f_{C,t}$) for the term in the corresponding cluster. These fields are used during query-cluster similarity computation and in fact, represent the centroids used for the selection of the best-clusters. Therefore, in our approach the centroid information is stored with, or embedded into document posting lists. In Figure 4.1 each posting list header contains the associated term, the number of posting list elements (pairs) associated with that term, number of clusters containing the term, and the posting list pointer (disk address). The posting list elements are of three types, *(cluster id, position of the next cluster)*, *(number of documents in the sub-posting list, average within-cluster term frequency)* and *(document id, term frequency)*. Note that, while the latter is a typical posting list element, the first two are called *skip-element* and *centroid-element*, respectively. In a posting list, the skip- and centroid-element along with succeeding typical elements (till the next skip-element) are called a *sub-posting list*.

In Figure 4.1, the posting list for t_6 includes documents from three clusters.

For the first two clusters, the centroid-elements simply store (1, 1) since the number of documents in cluster C_1, (C_2) is 1, as well as the average within-cluster term frequency. For the last cluster in this posting list, the centroid-element is (3, 3) since there are three documents in cluster (d_5, d_6, d_7) and the average within-cluster term frequency (as an integer) in the cluster is (5+4+1)/3 = 3.

An immediate benefit of this new inverted index structure is that, there is no need for a separate centroid index, and subsequently there is no need for an additional direct disk access time per query term for fetching the centroid IIS posting list (assuming that the latter would reside on disk). By embedding cluster information into the posting lists, any term in a cluster (or all of the terms) can be chosen as a centroid term and during the query processing its weight can be computed by using the methods described in Section 3.2.3.1. For simplicity, assume that all terms that appear in a cluster are used in the cluster centroids. In this case, the within-cluster term frequency of the term is required to compute the *tf* component of the term weighting schemes (e.g., CW2 and CW3 of Table 3.2). This value is approximately computed as the product of the values stored in the centroid-element in a sub-posting list (i.e., *sub-posting list length* $\times$ *average within-cluster term frequency*), as shown in the line 7 of Algorithm 5. Note that, instead of storing the actual $f_{C,t}$ value in the centroid-element, we prefer to store the average frequency value, and obtain the actual value by a multiplication. This is for the benefit of compression process (discussed in the next section), as smaller integers occupy less space during compression. We expect that using an approximate value instead of the actual $f_{C,t}$ in a cluster does not affect overall system effectiveness, which is justified by the experimental results. For the *idf* component of weighting schemes, the number of clusters including a term is required. Notice that, this information is captured in the CS-IIS header (see Figure 4.1).

Note that, we assume that cluster lengths (i.e., centroid normalization factors used in matching [97]) are pre-computed and stored just like document lengths for whichever term weighting scheme is used. During query processing, centroid term weights are normalized by using the pre-computed cluster lengths.

Algorithm 5 The ranking-query evaluation algorithm for incremental-CBR with CS-IIS

Input: Query Q, CS-IIS I, document lengths DL, cluster lengths CL, no. of best-clusters to be selected n_s, no. of best-documents to be selected d_s

```
1: Sort the terms t of Q in descending order of term weight w_{q,t}
2: for each term t in Q do
3:    Retrieve I_t from I
4:    // First pass over the posting list: selecting the best-clusters
5:    for each sub-posting list IS_t in I_t do
6:       Access the (Cid, address) and centroid-element in IS_t
7:       Compute w_{Cid,t} using centroid-element
8:       CAcc[Cid] ← CAcc[Cid] + w_{q,t} × w_{Cid,t}
9:       Go to the next skip-element pointed to by address
10:   Normalize nonzero CAcc entries using CL
11:   Select n_s clusters with highest CAcc scores into BestClus using a min-heap
12:   // Second pass over the posting list: selecting the best-documents
13:   for each sub-posting list IS_t in I_t do
14:      Access the skip-element (Cid, address) from IS_t
15:      if Cid ∈ BestClus then
16:         for each posting (d, f_{d,t}) in IS_t do
17:            Compute w_{d,t} using f_{d,t}
18:            DAcc[d] ← DAcc[d] + w_{q,t} × w_{d,t}
19:      else
20:         Go to the next skip-element pointed to by address
21: Normalize nonzero DAcc entries
22: Select d_s documents with highest DAcc scores using a min-heap
```

4.2.2 Incremental Cluster-Based Retrieval

In incremental-CBR, we determine the best-clusters by only accessing the cluster-skipping IIS. The basic heuristic is that, instead of determining the final best-clusters before ranking the documents in these clusters, as in the case of typical-CBR, we progress both processes in incremental fashion. In this new strategy, the query terms are processed in decreasing order according to their weights. For a given query, the posting list for the most important query term is brought to memory. In the first pass over its posting list, the *best-clusters-so-far* are determined using an appropriate centroid term weighting scheme (see Section 3.2.3.1) and similarity measure. Notice that, the information required for these schemes are available in the skip- and centroid-elements (as mentioned in the above section), so during the first pass it is sufficient to access *only* to those elements of

each sub-posting list. In the second pass, only those documents whose clusters fall into the *best-clusters-so-far* are considered, while the system skips the documents that are not in the best-clusters as before. The same is repeated for the next term in order (see Algorithm 5). Remarkably, during query processing only necessary elements of the CS-IIS are accessed in each pass. This is especially important for reducing the number of decoding operations in a compressed environment.

For instance, assume a query that contains the terms $\{t_4, t_6\}$ and the number of best-clusters (n_s) and number of best-documents (d_s) to be selected are 2. Further, assume that t_4 has a higher term weight than t_6 for this query (see Figure 4.2). Then, first the posting list of t_4 is fetched. In the first pass, the query processor reaches only the skip- and centroid-elements in the posting list and updates the cluster accumulator entries for C_1 and C_2. Let us assume that their similarity scores are (partially) computed as 0.65 and 0.75, respectively. Then, since the number of best-clusters to be selected is 2, these two clusters will be in *best-clusters-so-far*, and in the second pass the document accumulator entries for the documents in these clusters, namely, documents d_2, d_3, d_4 will be updated (say, as 0.1, 0.3, 0.7, respectively). Next, the posting list of t_6 is fetched. Let us assume that this updates cluster accumulator entries for clusters C_1, C_2 and C_3 with the additional values 0.20, 0.05 and 0.90, respectively. Now, the *best-clusters-so-far* includes C_1 and C_3 with scores 0.85 and 0.90 whereas C_2 with score 0.80 is out, and thus the documents from these two clusters are considered but sub-posting list for C_2 is skipped during the second pass. That is, the documents d_1, d_5, d_6 and d_7 will be updated (say, as 0.1, 0.5, 0.4 and 0.1, respectively). The highest-ranking two documents, d_4 and d_5, are returned as the query output.

In summary, the proposed incremental-CBR strategy with the CS-IIS file has two major advantages: First, embedding cluster information into the IIS and the incremental query evaluation method eliminate the need for a separate centroid IIS and hence disk access time to retrieve its posting lists. This means, in a memory-scarce environment where the index files are kept on disk, incremental-CBR achieves half of the number of direct disk accesses required by typical-CBR, and the same number of direct disk accesses required by FS. Second, cluster skipping and thus, decoding only relevant portions of CS-IIS during both stages of

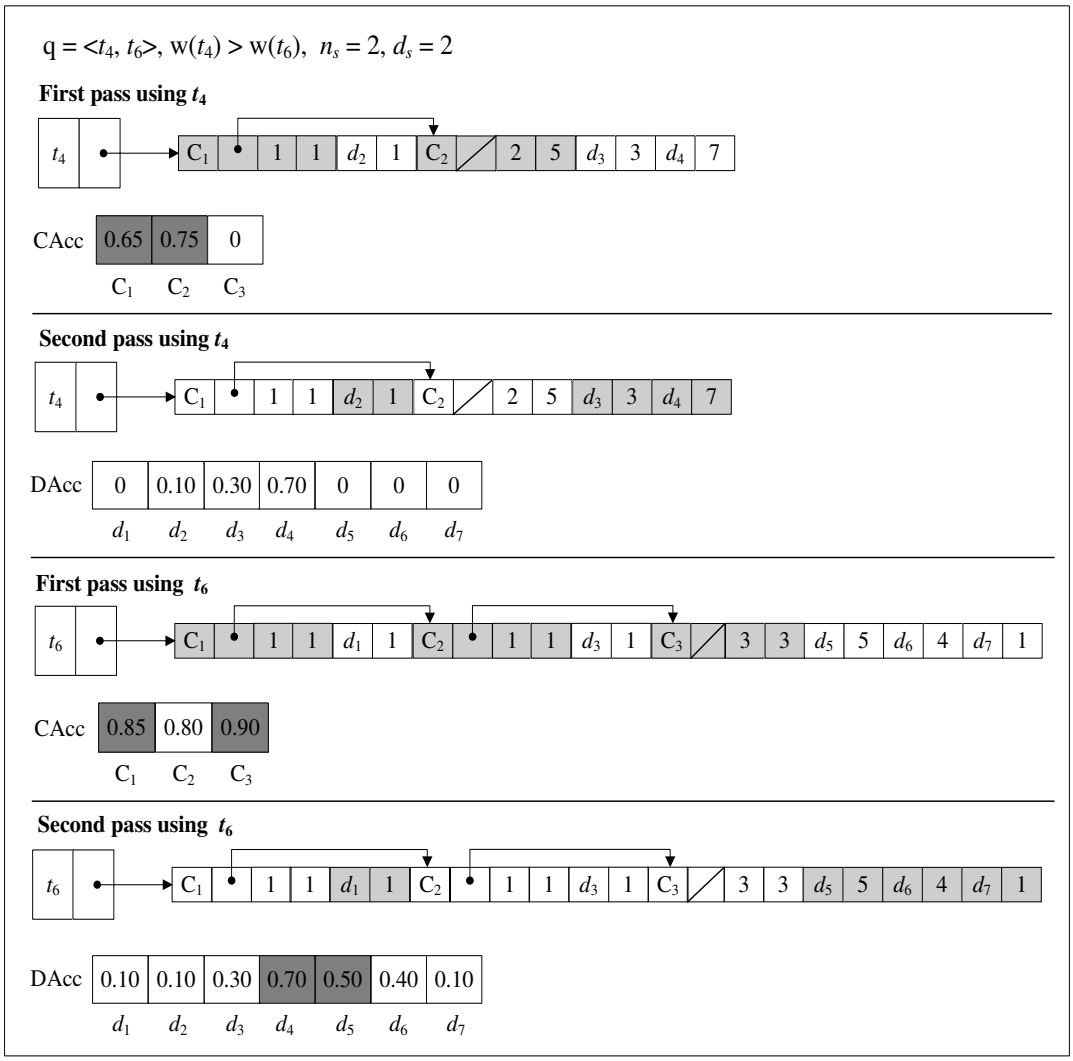


Figure 4.2: Example query processing using incremental-CBR strategy (accessed and decompressed list elements are shown with light gray, best documents and clusters are shown with dark gray).

query processing saves significant decompression overhead. This means improved in-memory query processing performance with respect to typical-CBR and FS. In the next section, we discuss how we handle the only overhead of CS-IIS, storage consumption due to newly added skip- and centroid-elements, by adapting the compression techniques in the literature 3.2.1.1.

4.3 Compression and Document ID Reassignment for CS-IIS

4.3.1 Compressing CS-IIS

As discussed before, the cluster-skipping IIS includes three types of elements in posting lists: i) the skip-elements in the form of (cluster id, position of the next cluster), ii) the centroid-elements in the form of (sub-posting list length, average $f_{C,t}$), and iii) the typical elements of type $(d, f_{d,t})$. For the compression of such a posting list, we consider three types of gaps: *c-gaps* between the cluster ids of two successive sub-posting lists, *a-gaps* between address fields (i.e., following the approach taken in [79]), and the typical *d-gaps* for document ids.

Example 4.1 Let us consider the posting list entry for t_3 of Figure 4.1, in which skip- and centroid-elements are shown in bold.

(1, add2) (2, 1) (1, 1) (2, 1) **(2, add3) (1, 1)** (4, 1) **(3, EOL) (1, 1)** (7, 1)

The list to be compressed will be represented as follows:

(1, add2) (2, 1) (1, 1) (1, 1) **(1, add3-add2) (1, 1)** (4, 1) **(1, EOL) (1, 1)** (7, 1)

Note that, the underlined fields are represented as gaps. End Of List (EOL) is represented by the smallest possible integer that can be compressed; i.e., 1.

There are two subtle issues regarding the above representation. Assume that d-gaps are encoded by using the Golomb code with the local Bernoulli model, which is a common practice in the literature [118]. In this case an appropriate way

of computing the Golomb parameter (b) is required, since the original formulation does not consider that documents in our CS-IIS are grouped together according to their clusters (i.e., into sub-posting lists) and the document id distribution probability must be revised to reflect this modification, as well. As a simple solution, for posting list I_t for term t we revise the previously given formula (Equation 3.3) as in Equation 4.1, assuming that the documents with term t is uniformly distributed among the clusters that appear in I_t. The number of clusters is assumed to be stored with the header of the IIS (see Figure 4.1).

$$b = 0.69 \times \frac{N}{f_t/(no.\ of\ clusters\ in\ I_t)} \tag{4.1}$$

The second important observation from the above representation is that, for the CS-IIS, the first document id in each sub-posting list per cluster (e.g., d_1, d_4 and d_7 in the above example) should be encoded as-is, which may significantly diminish the compression ratio. In the next section, we propose a remedy for this problem.

4.3.2 Document Id Reassignment

Document id reassignment is an emerging research topic that attempts to make document ids in a posting list as close as possible, so that the frequency of small d-gaps improves compression rates [102]. Here, we apply an apparently natural document id reassignment method: essentially, the documents in the same cluster are assigned consecutive ids, and the order among clusters is determined according to their creation order by the clustering algorithm. Similarly, the order of the documents in a cluster is determined by the order of entrance of these documents into the cluster. Notice that, a similar approach using k-means clustering algorithm is reported in [102] among many other techniques. In that work, it is also reported that some other techniques (as in [23]) can provide better compression rates (i.e., up to 10% smaller) with respect to a cluster-based scheme as described above. However, we prefer to use the cluster-based reassignment method, which can be amortized by and computed during the clustering process.

For CS-IIS, the expected benefit of document id reassignment is two fold: i) in each sub-posting list per cluster, the d-gaps between successive documents ids are reduced, and ii) more importantly, the id of the first document, which must be encoded as-is, in each sub-posting list can be reassigned a smaller value. Indeed, with a little main-memory consumption, it is possible to amplify the benefit mentioned in (ii) significantly. In each cluster, documents are assigned a *real id*, which is determined as described above, and a *virtual id*, which starts from 1 and increments by 1, just to be used for the compression purposes. During compression, virtual ids are compressed, so that each sub-posting list would start with a considerably smaller id than the original one. During query processing, an array is kept in main memory to store prefix sum of cluster sizes, so-called, *size-sum array*. Whenever a document id field is decoded, the decoded virtual id is added to the prefix sum value stored for this document's cluster (which is already known, since decoding starts from the skip-element per sub-posting list) in the size-sum array to obtain the real id, and corresponding correct document accumulator is updated for this real id.

Example 4.2 Assume that cluster C_1 includes two documents and cluster C_2 includes three documents. The documents from C_1 and C_2 will be assigned to real ids 1, 2, and 3, 4, 5, respectively. The virtual ids are also 1 and 2 for C_1, but 1, 2 and 3 for C_2. The size-sum array will store 0 for C_1, $0 + sizeof(C_1) = 0 + 2 = 2$ for C_2. During query processing, if a document id in C_2's sub-posting list is decoded as 2, it will be added to size-sum array value for C_2, which is also 2, to obtain the real id as 4.

Note that, number of clusters would be smaller than the number of documents in the order of magnitudes, so that storing size-sum array in the memory is not a major problem. Furthermore, the array can be kept in the shared memory and accessed by several query processing threads at the same time; i.e., it is *query invariant*. Finally, if the Golomb code is employed for encoding d-gaps, the b parameter should be further revised. In particular, we refine it as Equation 4.2, since the virtual documents ids in each sub-posting list can range from 1 to "average cluster size" on the average.

$$b = 0.69 \times \frac{average\ cluster\ size}{f_t/(no.\ of\ clusters\ in\ I_t)} \tag{4.2}$$

As another alternative, we can define a dedicated b value to compress each sub-posting list separately (Equation 4.3). Note that, cluster size C_i can be easily computed from the size-sum array as the difference of array entries for $i+1$ and i, without requiring an extra data structure. The number of occurrences of t in C_i (f_{t,C_i}) is captured in the centroid-element of IS_i (i.e., sub-posting list length) and will be decoded immediately before the decoding of d-gaps start. In Section 4.5, we evaluate and compare the storage figures for various compression schemes and parameters.

$$b = 0.69 \times \frac{size(C_i)}{f_{t,C_i}} \tag{4.3}$$

4.4 Experimental Environment

4.4.1 Datasets and Clustering Structure

In the experiments, three datasets are used. The *Financial Times* collection (1991-1994) of TREC [108] Disk 4, referred to as the FT dataset, and the *AQUAINT* corpus of English News Text, referred to as the AQUAINT dataset, are used in previous TREC conferences and include the actual data, query topics and relevance judgments. These two datasets are also used in the experiments of Chapter 3 and their features are repeated here for easy referencing. As a third dataset, we obtained the crawl data from the Stanford WebBase Project Repository [114]. This latter dataset, referred to as the WEBBASE, includes pages collected from the US government Web sites during the first quarter of 2007. As there are no query topics and relevance judgments for this dataset, it is solely used for evaluating query processing efficiency. During the indexing stage, we eliminated English stop-words, and indexed the remaining words, and no stemming is used. For the WEBBASE dataset, the words that appear in only one document are also removed, as the Web pages include a high number of mistyped

Table 4.1: Characteristics of the datasets

Dataset	Size on disk	No. of documents(N)	No. of terms(n)	No. of clusters	No. of (d, $f_{d,t}$) pairs	Avg. no. of docs/clusters
FT	564 MB (text)	210,158	229,748	1,640	29,545,234	128
AQUAINT	3 GB (text)	1,033,461	776,820	5,163	170,004,786	200
WEBBASE	140 GB (HTML)	4,293,638	4,290,816	13,742	790,291,775	312

words. In Table 4.1, we provide statistics for the datasets and the indexing results. Notice that, the original WEBBASE dataset spans more than 140 GB on disk in HTML. After preprocessing and removing all HTML tags, scripts, white spaces, etc. the pure text on disk (tagged in TREC style) takes 27 GB.

The datasets are clustered using C^3M algorithm [42] in partitioning mode as discussed in Chapter 3, which yields 1,640, 5,163 and 13,742 clusters for the FT, AQUAINT and WEBBASE datasets, respectively. An important parameter for CBR is the number of best-clusters. In Chapter 3 it has been reported that the effectiveness increases up to a certain n_s value, after this (saturation) point, the retrieval effectiveness remains the same or improves very slowly for increasing n_s values. This saturation point is found around 10 to 20% in the literature [42, 95, p. 376]. Therefore, in the retrieval experiments reported in Section 4.5, we use 10% of the total number of clusters as the number of best-clusters to be selected (i.e., n_s is 164, 516 and 1,374 for the corresponding datasets). In this study, we provide results for retrieving top-1000 documents; i.e., number of best-documents to be selected, d_s, is 1000.

The clustering of the largest dataset (WEBBASE) takes around twelve hours using a rather out-dated implementation of C^3M algorithm. Once the clustering is completed, creating the typical IIS and CS-IIS takes almost equal times, which is around a few hours for this dataset, by again using an unoptimized implementation. Nevertheless, any partitioning type clustering algorithm could be used in our setup, given that the algorithm can provide reasonable effectiveness by accessing a relatively small percentage of all clusters.

Table 4.2: Query sets' summary information

Dataset & Query Sets	No. of queries	Avg. no. of relevant documents	Query type	Avg. no. of terms	Min no. of terms	Max no. of terms
FT, Qset1	47	31.8	Qshort	2.5	1	4
			Qmedium	10.8	4	30
FT, Qset2	49	38.1	Qshort	2.4	1	3
			Qmedium	8.2	2	19
			Qlong	190.0	13	612
FT, Qset3	49	33.4	Qshort	2.4	1	3
			Qmedium	7.3	3	19
AQUAINT, Qset1	50	131.2	Qshort	2.5	1	4
			Qmedium	9.4	4	20
WEBBASE, Qset1	50,000	N/A	Qshort	2.3	1	9

4.4.2 Query Sets and Query Matching

For the FT dataset, we used three different query sets along with their relevance judgments that are obtained from the TREC Web site [108]. The three query sets, referred to as $Qset1, Qset2$ and $Qset3$, include TREC queries 300-350, 351-400 and 401-450, respectively. Note that, the relevance judgments for some of the queries in these sets refer to the documents that are from datasets other than the ones used in this work. Such irrelevant judgments are eliminated, and for each query set we produce a relevance judgment file, which includes only the documents from the FT dataset. A few of the queries do not have any relevant documents, and they are discarded from the query sets. Table 4.2 shows the remaining number of queries for each query set of FT. For the AQUAINT dataset, we used the topics and judgments used for TREC 2005 robust track. Finally, for the WEBBASE dataset, the efficiency task topics of TREC 2005 terabyte track are employed. Note that, this query set have been used on top of the TREC GOV2 dataset, which also includes Web data from the "gov" domain. Since the WEBBASE collection also captures the same domain, we presume that this query set is a reasonable choice for efficiency evaluation with WEBBASE.

In the experiments, we used two different types of queries, namely $Qshort$ and $Qmedium$ that are obtained from the query sets discussed above. $Qshort$ queries include TREC query titles, and $Qmedium$ queries include both titles and descriptions. For one of the FT query sets (FT-$Qset2$), we also formed a third query type, $Qlong$, which is created from the top retrieved document of each

Qmedium query in this query set. Our query sets cover a wide spectrum from very short Web-style queries (the *Qshort* case) to extremely long ones (the *Qlong* case). Notice that, the latter type of queries can capture the case where a user likes to retrieve similar documents to a particular document and the document itself serves as a query. Table 4.2 provides query sets' summary information.

In the following experiments, the document term weights are assigned using the *tf-idf* formula. The cosine function is employed for both query-cluster and query-document matching. Please refer to Section 3.2.1 for further details.

4.4.3 Cluster Centroids and Centroid Term Weighting

For the cluster centroids, we follow the practice in Chapter 3 and use all cluster member documents' terms as centroid terms. Note that, this choice of centroids also enables us being independent of a particular centroid term selection method. Nevertheless, it is possible to apply other centroid term selection schemes in our framework as well. The experiments employ the three centroid weighting schemes as described in Table 3.2. Recall that, the information stored in the enhanced CS-IIS file is adequate to compute all three schemes, as mentioned in Section 4.3.

4.5 Experimental Results

The experiments are conducted on a Pentium Core2 Duo 3.0 GHz PC with 2GB memory and 64-bit Linux operating system. All IR strategies are implemented using the C programming language and source codes are available on our Web site. Implementations of the IR strategies are tuned to optimize query processing phase for which we measure the efficiency in the following experiments. In particular, a min heap is used to select best-clusters and best-documents from the corresponding accumulators as recommended in previous works [118]. Unless stated otherwise, we assume that the posting list per query term is read into main-memory, processed and then discarded; i.e., more than one term's posting list is not memory resident simultaneously. The document lengths and cluster lengths are pre-computed.

In what follows, we first compare the effectiveness figures of the incremental-CBR strategy with those of the FS and typical-CBR, to demonstrate that the new strategy does not deteriorate the quality of query results. Next, we focus on the efficiency of the proposed strategy and show that incremental-CBR is better than FS in total query processing performance (involving in-memory evaluation and disk accesses) with a reasonable overhead in the storage requirements. Finally, we show that incremental-CBR is superior than not only a basic implementation of FS but a faster approach that employs the "continue" pruning strategy along with a skip embedded IIS, as described in [79].

4.5.1 Effectiveness Experiments

In this section, we compare three IR strategies, FS, typical-CBR, and incremental-CBR with CS-IIS. To evaluate the effectiveness of the proposed strategy, the top 1000 (i.e., d_s= 1000) documents are retrieved for each of the query sets. The effectiveness results are presented by using a single mean average precision (MAP) value for each of the experiments. All MAP scores are computed using the `treceval` software [108] and the result files are available at our Web site[1].

The effectiveness results obtained for FS experiments are compared to those obtained by using a publicly available search engine, Zettair [121], to verify the validity of our findings and robustness of our implementation. The indexing and querying stages with Zettair are achieved under almost the same conditions as our own implementations. During indexing, no stemming is used. In query processing, the same stop-word list as we use in our system is provided to Zettair and the cosine similarity measure is chosen. For each dataset, *Qshort* and *Qmedium* query types are evaluated by retrieving top-1000 results. We found that, in almost all experiments our MAP values are slightly better than those of Zettair, which validates our implementation.

The first observation that can be deduced by a quick glance over Table 4.3 is that for each query set and type, all MAP values are very close to each other

[1] `http://www.cs.bilkent.edu.tr/~ismaila/PhD/sources.htm`

Table 4.3: MAP values for retrieval strategies ($n_s = 164$ for FT, $n_s = 516$ for AQUAINT, $d_s = 1000$)

Datasets & Query Sets	Query Type	FS	Typical-CBR			Incremental-CBR		
			CW1	CW2	CW3	CW1	CW2	CW3
FT, *Qset*1	*Qshort*	0.161	0.162	**0.168**	0.154	0.163	0.167	0.166
	Qmedium	0.152	**0.173**	0.148	0.143	0.158	0.153	0.155
FT, *Qset*2	*Qshort*	0.107	0.126	0.109	0.102	**0.131**	0.110	0.110
	Qmedium	0.122	0.134	0.121	0.113	**0.137**	0.120	0.120
	Qlong	**0.124**	0.113	0.114	0.109	0.119	0.120	0.119
FT, *Qset*3	*Qshort*	**0.154**	0.142	0.144	0.131	0.134	0.150	0.147
	Qmedium	**0.170**	0.150	0.166	0.123	0.159	0.161	0.142
AQUAINT, *Qset*1	*Qshort*	**0.091**	0.046	0.081	0.071	0.047	0.081	0.077
	Qmedium	**0.100**	0.048	0.089	0.074	0.057	0.090	0.081

(the best ones are shown in bold). Thus, it is hard to claim that one single strategy totally outperforms the others. Still, the results demonstrate that CBR is a worthwhile alternative to FS for accessing large document collections.

From the above results it is clear that the proposed strategy has no adverse effect on CBR effectiveness and in particular cases, it can even improve effectiveness. In particular, Table 4.3 reveals that incremental-CBR strategy is better than the typical-CBR for the majority of the cases, although the absolute MAP improvement is rather marginal. For *Qshort* and *Qmedium* query types of *Qset*2 on the FT dataset, the incremental-CBR strategy yields the best effectiveness figures, outperforming both FS and typical-CBR. Another interesting observation is that for the CBR strategies, CW1 and CW2 are the most promising centroid term-weighting schemes.

We conduct a series of matched pair t-tests to determine whether incremental and typical CBR strategies with CW1, CW2, and CW3 are as effective as FS. The null hypotheses in this case would be that the effectiveness of each of these methods is as good as FS and the alternative is that they are not as good. For this purpose, we examine the performance differences of these two approaches provided in Table 4.3. Note that we are performing one-sided t-tests so we would divide the two-sided p-value by 2. Since we are also performing 6 hypothesis tests we perform a Bonferonni correction by multiplying each p-value by 6. Thus, combining the two adjustments, we end up multiplying each two sided p-value by 3. So a significant result would be a p-value that is less than $0.05/3 = 0.016$. Each

difference is the average of the CBR method subtracted from the full search (FS) for each query type. Since the average differences are negative, on the average, FS outperforms each cluster method in terms of MAP. However, the only significant difference (based on p-values) is the difference between CW3 for typical-CBR and FS ($p < 0.01$). In this case, FS significantly outperforms typical-CBR with CW3. However, in the other tests there is a lack of evidence that FS significantly outperforms CBR. Since CBR with CW1 and CW2 outperform FS for some query types, CBR has the potential of being as effective as FS.

Finally, it should be emphasized that the incremental-CBR strategy with CS-IIS is not at all intended to improve effectiveness of CBR, but it aims to improve efficiency without deteriorating the effectiveness of the typical-CBR while providing compatible effectiveness with FS. Recall that, there are recent proposals to improve CBR effectiveness [73] that can obviously be applied in our framework, as well.

4.5.2 Efficiency Experiments

In the following experiments, we compare incremental-CBR to only FS, as our goal in this chapter is to propose a CBR strategy that is more efficient than FS especially when the best-cluster selection cost is also involved for the former one. For the efficiency experiments, we report the results obtained by using all three datasets shown in Table 4.1 and corresponding query sets. However, to shorten the discussion, we only use $Qset2$ for the FT dataset, for which the effectiveness of incremental-CBR also peaks.

4.5.2.1 Storage Efficiency

In Table 4.4, we provide the compressed file sizes for the evaluated IR strategies. In particular, the term frequency values in typical IIS and both fields of the skip- and centroid-elements in CS-IIS are encoded with Elias-γ code. The values d-gaps are encoded by using Elias-γ and Golomb codes in separate experiments. This is due to the observation that, one of the schemes, namely the Golomb code,

Table 4.4: File sizes (in MBs) of IIS (for FS) and CS-IIS (for Incremental-CBR), Raw: no compression, LB: local Bernoulli model, OrgID: original doc ids, ReID: reassigned doc ids

Dataset	FS					Incremental-CBR				
	Raw	Golomb(LB)		Elias-γ		Raw	Golomb(LB)		Elias-γ	
	IIS	OrgID	ReID	OrgID	ReID	IIS	OrgID	ReID	OrgID	ReID
FT	225	34	33	44	**43**	343	84	45	105	**50** (14%>FS)
AQUAINT	1,360	211	209	236	**216**	1,900	520	254	602	**250** (16%>FS)
WEBBASE	6,322	1,076	1,079	767	**770**	7,362	2,315	968	1,745	**844** (10%>FS)

appears to be unaffected from the document id reassignment methods for typical IIS.

Table 4.4 reveals that for the FT and AQUAINT datasets, when the original documents ids in the collections are used, the best compression rates for typical index files are achieved by using the Golomb code with LB (using Equation 3.3). For the WEBBASE dataset, however, Elias-γ performs better (i.e., 767 vs. 1,076 MB). We attribute this to the observation that in the latter dataset, which is yielded by a crawling session, the original document ids are sorted in URL order that exhibits strong locality [101]. On the other hand, the Golomb code is rather insensitive to such locality and performs best on random distributions [23]. This phenomenon is strongly emphasized by the experiments with the reassigned document ids and further discussed below. Nevertheless, for the WEBBASE dataset, the typical IIS size drops from 6,322 MB to 1,076 MB (17%) and 767 MB (13%) with Elias-γ and Golomb (with the LB model using Equation 4.1) schemes, respectively. The compressed IIS sizes also correspond to only 4% and 3% of the uncompressed text document collection (27 GB) for respective cases. This conforms to the results reported in other works in the literature [118]. On the other hand, it is seen that the compression gains on CS-IIS by using original document ids are not as good, and for WEBBASE dataset, the compressed file sizes are 31% and 24% of the uncompressed index using the two compression schemes. However, at this point, the potential of document id reassignment, which is naturally applicable for CS-IIS, has not been exploited yet.

Next, we applied the document id reassignment method mentioned in Section 4.3.2, so that documents in each cluster have consecutive ids. For this experiment, we first discuss the results when the Golomb code is used to encode d-gaps. Note that, the b parameter for LB is set as in Equation 3.3 for typical IIS, whereas the enhanced formula derived in Section 4.3.2 is (Equation 4.3) employed for CS-IIS, to reflect the distribution of sub-posting lists as accurate as possible. Remarkably, the Golomb code with LB provides almost no improvement for the typical IIS, whereas CS-IIS highly benefits from the reassignment. For instance, the size of CS-IIS file for WEBBASE dataset drops from 2,315 to 968 MB, a reduction of more than 50%. This is even less than the compressed size of typical IIS (1,079 MB) for the corresponding case. As it is mentioned before, the insensitivity of typical IIS for reassigned ids is caused from the characteristics of the Golomb code, which cannot exploit the locality (i.e., it should still use the same b parameter for LB after reassignment). In particular, for FT and AQUAINT datasets the reductions in the compressed index sizes are at most 3%, hard to call as an improvement. For WEBBASE, there is even a slight increase (0.3%) in the index size. On the other hand, after reassignment, the CS-IIS allows to use an enhanced b parameter (Equation 4.3) and benefits from the reassignment procedure even when the Golomb code is used.

For the sake of fairness, we repeated the experiments with reassigned ids and by encoding d-gaps with the Elias-γ method. In this case, as Table 4.4 demonstrates, the typical IIS also obtains some gains from document id reassignment, but the gains are still less impressive in comparison to CS-IIS. Noticeably, the storage space used for compressed index files of FT and AQUAINT drops by 2% and 9%, respectively. For WEBBASE, there is no improvement on the index size, but again a slight increase is observed. This is due to the fact that, the originally URL-ordered ids for this dataset provides quite strong locality, and the reassignment based on clustering does not further improve the compression rate (a result also shown in [101]). To validate this claim, we assigned random ids to documents in the WEBBASE dataset and repeated the compression experiments. In this case, the compressed index sizes are 1,132 and 1,473 MB for the Golomb and Elias-γ methods, respectively. These results support our claims, in

that, i) if the original document ids are not sorted in URL order, the Golomb code with LB would provide better compression rates (as in the cases of FT and AQUAINT) with respect to Elias-γ, ii) the Golomb code is rather not sensitive to any locality (the file sizes for random and URL-sorted experiments are very close, 1,132 and 1,076 MB, respectively) whereas Elias-γ is just the reverse (i.e., the index size drops from 1,473 to 776 MB), and iii) sorting by URL order provides a very good d-gap distribution as shown by the results of Elias-γ, and the typical IIS size cannot be reduced by further reassignment. In contrast, CS-IIS still significantly benefits from id reassignment; i.e., yielding reductions of more than 50% in size For instance, by using the Elias-γ encoding method, the CS-IIS file for WEBBASE only takes 844 MB on disk, which is only 10% larger than the typical IIS for corresponding case (770 MB). This is a striking result for the space utilization of CS-IIS that is obtained by using a cluster-based document id reassignment technique which is a natural advantage of our framework.

Recall that, the document reassignment method for CS-IIS employs virtual ids instead of real ids in the sub-posting lists, to encode the first document of each sub-posting list more efficiently (see Section 4.3.2). We devised a separate experiment to evaluate the performance of this heuristic. For the WEBBASE dataset, we simply reassigned documents ids. In this case, the first document id of each posting list, which should be compressed as-is, takes 330 MB and 232 MB of the resulting CS-IIS file, for the Elias-γ and the Golomb code with LB (using Equation 3.3), respectively. Next, we applied the optimization of Section 4.3.2 (i.e., virtual ids are assigned within each cluster to reduce the actual value of first document ids in sub-posting lists). In this case, only 100 MB and 76 MB of CS-IIS is devoted to first ids, again for the Elias-γ and the Golomb code with LB, respectively. For the latter scheme, the b parameter for Golomb is now set as in Equation 4.3, which is a unique opportunity allowed by CS-IIS. Notice that, for both compression schemes, our optimization reduces the space used for first ids to almost one third of the original space. Moreover, our formulation for the b parameter allows the Golomb code to provide much better compression ratio with respect to Elias-γ; i.e., leading to a further 24% reduction in size. This experiment shows that, efficient compression of the first document id in each sub-posting list

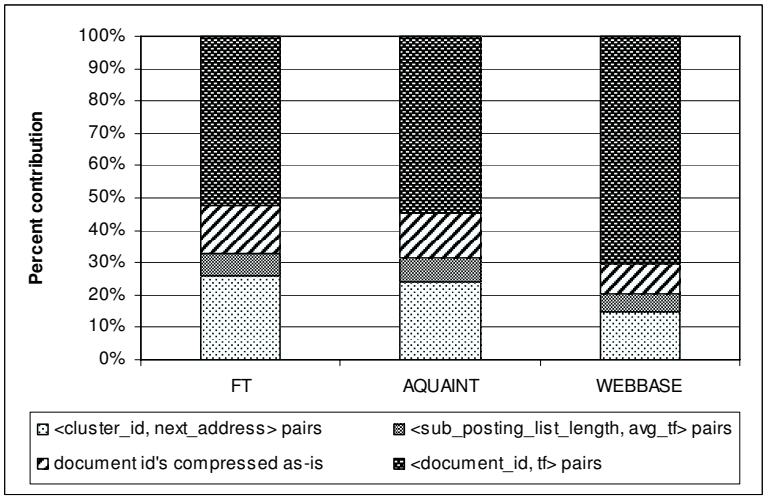


Figure 4.3: Contribution of CS-IIS posting list elements to compressed file sizes for the three datasets.

of CS-IIS is important for the overall compression efficiency, and the heuristic outlined in this study provides significant gains. Therefore, in all reassigned id experiments for CS-IIS (as reported in Table 4.4), the first document ids are always encoded with the Golomb code, regardless of the schemes the remaining d-gaps are compressed. Note that, in this heuristic, the size-sum array (of size number of clusters) takes only a few KBs of in-memory space even for WEBBASE, which is a negligible cost.

In summary, by using a cluster-based id reassignment approach, both the Golomb coding with the LB model and Elias-γ schemes prove to be quite successful for compressing CS-IIS. Remarkably, by using the Elias-γ scheme, the additional cost of storing CS-IIS, with respect to typical IIS, is at most 16% (see the last column of Table 4.4). In the remaining experiments, we use the compressed typical IIS and CS-IIS files that are obtained by the id reassignment and the Elias-γ encoding for d-gaps; i.e., those shown as bold in Table 4.4.

In Figure 4.3, we provide the percentage of storage for each field in the compressed CS-IIS file (for the file sizes in the last column of Table 4.4). Considering the figure, we realize that for WEBBASE, 70% of the file is used to store actual (document id, tf) pairs, whereas 15% is used for the skip-elements (i.e., in the

form of (cluster id, next address)), and 5% is used for the centroid-elements (i.e., in the form of (sub-posting list length, avg. $f_{C,t}$)). Since each sub-posting list encodes its first document as-is, a considerable fraction of the file (around 10%) is used for this purpose. Notice that, while our extra posting list elements cause 30% of the overall cost in CS-IIS, they also allow document id reassignment to be more efficient, and thus the overall size remains within an acceptable margin of typical IIS. Figure 4.3 also shows that the percentage of the additional storage in CS-IIS reduces as the dataset gets larger; i.e. 50%, 45%, and 30% for FT, AQUAINT, and WEBBASE, respectively. Remarkably, these percentages are not necessarily reflected to CS-IIS file size as increments, as discussed above.

Finally, the compression process takes the same time for corresponding cases by using our own implementations, ranging from a few minutes (for FT) to an hour (for WEBBASE).

4.5.2.2 Query Processing Time Efficiency

In Table 4.5, we report average CPU (in-memory) processing times per query, as well as the average number of decode operations (i.e., total number of Elias-γ and Golomb decode operations). The experimental results are provided for CW1 and CW2; the CW3 case is omitted since its efficiency figures are similar to that of CW1.

The results reveal that incremental-CBR decompresses significantly smaller number of elements compared to FS. This is caused by the fact that the former decompresses only relevant portions of a posting list, whereas FS, of course, must decode the entire posting list for a query term (note that, in Section 4.5.2.3, we also discuss a skipping-based pruning technique for FS, as discussed in [79]). For CW1, the savings of the incremental-CBR in terms of number of decode operations are more emphasized, ranging from 58% to 80% of the decode operations by FS. For CW2, incremental-CBR decodes more elements, but still the number of decoded elements is almost half of the FS case. These savings are reflected to time figures rather conservatively, especially for shorter queries. The time savings improve as the queries become longer (e.g., for AQUAINT the savings

Table 4.5: Efficiency comparison of FS and Incremental-CBR (times in ms)

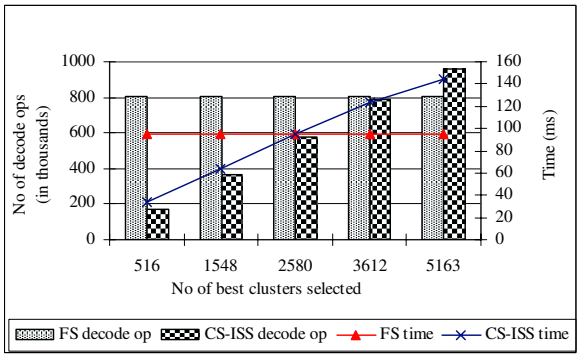
Datasets & Query Sets	Query Type	Avg. time & no. of decode op.	FS	Incremental-CBR		Imp. over FS	
				CW1	CW2	CW1	CW2
FT, *Qset*2	*Qshort*	Exe. time	5	3	4	40	20
		Decode op.	19,524	8,212	11,614	58	41
	Qmedium	Exe. time	16	7	9	56	44
		Decode op.	98,832	36,701	51,772	63	48
	Qlong	Exe. time	389	144	222	63	43
		Decode op.	3,627,468	1,091,212	2,079,408	70	43
AQUAINT, *Qset*1	*Qshort*	Exe. time	27	15	19	44	30
		Decode op.	162,824	37,860	73,249	77	55
	Qmedium	Exe. time	95	34	48	64	49
		Decode op.	802,740	172,415	313,291	79	61
WEBBASE, *Qset*1	*Qshort*	Exe. time	66	36	57	45	14
		Decode op.	432,238	87,289	318,431	80	26

are 44% (30%) and 64% (49%) for *Qshort* and *Qmedium* using CW1 (CW2), respectively). If we assume that posting lists are kept in the main memory (due to OS caching and large memories), then these savings become final execution time improvements.

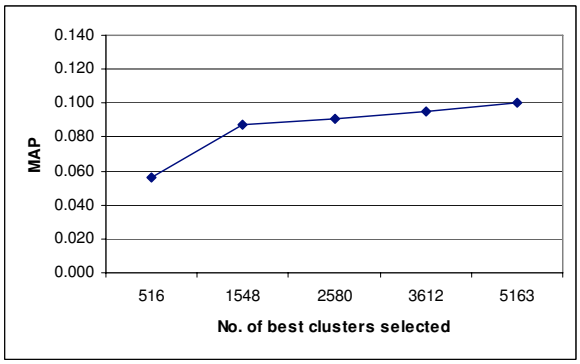
Note that, savings in time are not directly proportional to saving in the number of decode operations, because the incremental-CBR strategy with CS-IIS has also some overheads, such as jumping to the next bit position to be decompressed and selecting the best-clusters from the cluster accumulators for each query term.

In Figure 4.4(a), we plot the number of best-clusters selected vs. average number of decode operations (shown on the left y-axis of the plot) and average CPU query processing time (shown on the right y-axis of the plot) for FS and incremental-CBR, for *Qmedium* using CW1 centroid weighting scheme and the AQUAINT dataset. At the extreme point, all clusters are selected and incremental-CBR degenerates into FS. The number of decode operations realized by incremental-CBR and execution time is lower than that by FS until more than 50% of clusters (i.e., greater than 2580) are selected. Nevertheless, in practical CBR systems, the number of best-clusters to be selected is a relatively small percentage of the total number of clusters [42, 95].

In Figure 4.4(b), we plot the variation of the number of best-clusters selected vs. effectiveness. Note that, after 30% of clusters are selected as best-clusters, the MAP figures change slightly. Thus, for AQUAINT dataset it is possible to set



(a)



(b)

Figure 4.4: Effects of the selected best cluster number on (a) processing time and decode operation number, (b) effectiveness (for *Qmedium* using CW1 on AQUAINT dataset).

Table 4.6: Average size of fetched posting lists per query (all in KBs)

Dataset & query set	Query type	FS	Incremental-CBR	Overhead over FS(%)
FT, *Qset*2	*Qshort*	10.54	12.56	19
	Qmedium	51.09	60.78	19
	Qlong	1670.77	1962.12	17
AQUAINT, *Qset*1	*Qshort*	73.48	83.55	14
	Qmedium	345.64	391.26	13
WEBBASE, *Qset*1	*Qshort*	147.96	157.75	7

best-clusters as 30% of all clusters (i.e., 1548). Note that, even for this case, both the number of decompression operations and execution time are still significantly less than those for FS (see Figure 4.4(a)). For the sake of uniformity, we keep best-clusters as 10% throughout the experiments.

In Table 4.6 we provide the average size of posting lists fetched from the disk during query processing. Both FS and incremental-CBR make only one direct access per query term, assuming that the entire list for a term is fecthed at once. As expected, the incremental-CBR fetches slightly longer posting lists with respect to FS (due to the storage overhead of skip and centroid- elements). Note that, the increase in the posting sizes remains marginal and does not exceed 20%.

We expect that the cost of these longer sequential accesses would be compensated by the in-memory improvements in decoding times. For instance, assume a (rather slow) disk with the transfer rate 10 MB/s. In this case, the

additional sequential read time cost of CS-IIS with respect to FS for processing a query in *Qshort* set of WEBBASE would be around only $\approx$ 1 ms (i.e., (157.75 − 147.96)KB/10MB/s). For this latter case, FS takes 66 ms in CPU whereas incremental-CBR takes 36 and 57 ms for CW1 and CW2 cases, respectively (see Table 4.5). Clearly, even with a slow disk, in-memory time improvements are far larger than the disk read overhead (i.e., 30 ms (for CW1) and 9 ms (for CW2) vs. 1 ms). Thus, as long as the number of clusters is significantly less than the number of documents, which is a reasonable assumption, our approach would be feasible. Furthermore, assuming that all or most of the posting lists are kept in the main memory, which is the case for some Web search engines, our significant performance gains obtained during in-memory query processing become the conclusive improvements.

4.5.2.3 Experiments with FS using the Continue Strategy and Skipping IIS

We also compare our method with a more efficient FS approach using another pruning technique in the literature. In particular, since our approach is inspired from an earlier work that enriches the typical inverted index with skip elements [79], it seems to be a natural choice to implement it and compare with our incremental-CBR approach.

In [79], a posting list has a number of skip-elements each followed by a constant-sized block of typical elements. A skip-element has two components: the smallest document id in the following block and pointer to the next skip-element (and block). This was shown to be very efficient for conjunctive Boolean queries in a compressed environment. In particular, after the first posting list is processed, a candidate set of document id's are obtained, which are looked for in the other lists. Obviously, while searching to see whether a document is in a particular block, skip-elements are very useful: if the document id at hand is greater than the current skip-element and less than the next one, this block is decompressed; otherwise search process jumps to the next skip-element without redundantly decompressing the block. Note that, this technique is impossible to

be used as-is with the ranking-queries, since there is no set of candidate documents as in the Boolean case. Therefore, quit and continue pruning strategies are accompanied with ranking-query evaluation to allow the skipping inverted index to be used. Since the effectiveness figures of continue is quite close to the FS without any pruning (referred to as *typical FS* below), we prefer to use the continue strategy in this work.

In the continue strategy, the query processor is to allowed to update only a limited number k of accumulators. Until this limit is reached, it decodes the entire posting list for each query term, just like typical FS. After this limit is reached, the non-zero accumulators that are updated up to this time serve as the candidate document ids in the Boolean case and are the only accumulators that can be updated. Thus, it is possible to use skip elements and avoid decompressing blocks that do not include any documents with corresponding non-zero accumulators. We refer to this strategy as *skipping* FS.

In [79], each posting list can have different number of skip elements, according to the size of the posting list and the candidate document set [79, p. 363]. It is also stated that the continue strategy can achieve comparable or better effectiveness figures even when 1% of total accumulators are allowed for update. A good choice while constructing the skipping inverted index is assuming that the same k value represents the number of candidate documents for the queries.

In this section, we use AQUAINT, the largest dataset with relevance judgments for the experiments. Since this collection includes around 1M documents, k is set to 10,000 (i.e., 1% of the total document number). For the same k value, a skipping IIS is constructed in exactly the same way as described in [79]. The resulting index file takes 279 MB, which is 18% larger than the IIS with no skips (i.e., 236 MB, as shown in Table 4.4). In Figure 4.5, the MAP figures using this IIS file and varying number of accumulators (k) is shown. As expected, the effectiveness figures at $k = 10$K is as good as the effectiveness score when all 1M accumulators are available; i.e., as in typical FS.

In Figure 4.6, CPU execution times for skipping FS strategy with varying

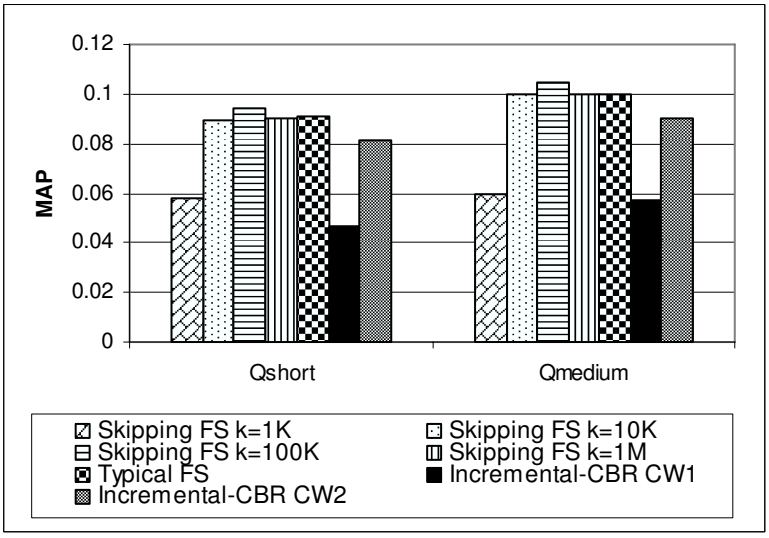


Figure 4.5: Effectiveness of skipping FS versus number of accumulators.

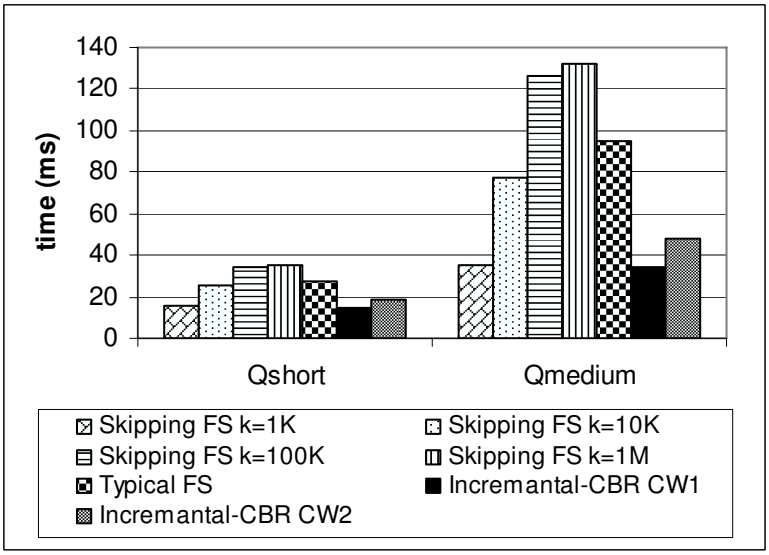


Figure 4.6: Query processing time of IR strategies.

Table 4.7: Number of decompression operations for skipping FS (with varying number of accumulators), typical FS and incremental-CBR

Query type	Skipping FS with continue strategy				FS	Incremental-CBR	
	k=1K	k=10K	k=100K	k=1M		CW1	CW2
Qshort	40,377	106,150	190,556	190,635	162,824	**37,860**	73,249
Qmedium	123,777	408,601	886,858	930,714	802,740	**172,415**	313,291

number of accumulators are reported (results for typical FS and incremental-CBR with CW1 and CW2 are also repeated from Table 4.5 for easy comparison). Clearly, skipping FS improves time performance of typical FS, up to 41% for *Qshort* and 63% for *Qmedium* when k = 1K. However, for this case, MAP scores also decrease. For k = 10K case, the improvements of skipping FS are 7% and 19% for *Qshort* and *Qmedium*, respectively. Nevertheless, the performance of incremental-CBR (with both CW1 and CW2) still remains to be superior. In Table 4.7, we report the number of decompression operations for the corresponding cases. Again, skipping FS improves over typical-FS, but cannot catch the incremental-CBR, for k = 10K case.

Finally note that, dynamic pruning techniques such as the one described above can also be applied to both typical- and incremental-CBR. For instance, during the best-documents selection stage of typical-CBR, it is possible to embed skipping FS approach. Similarly, the skipping strategy in this section can also be embedded into the sub-posting lists of CS-IIS (i.e., to provide another level of skipping in our approach). That is, many pruning techniques (as discussed in the next section) that can improve FS can also improve the CBR strategies. Integrating additional pruning techniques to typical- and incremental-CBR are beyond the scope of this thesis and left as future work.

4.5.2.4 Summary of the Results

In the experiments, we use various collections and multiple TREC query sets. These datasets constitute the largest collections used for document clustering and CBR. The experiments show that the incremental-CBR strategy with CS-IIS provides significant efficiency improvements while yielding comparable (or sometimes better) effectiveness figures. Our CPU query processing time efficiency

gains with respect to FS are impressive and up to 45% for Web style queries. The increment in the size of compressed posting lists is marginal. This overhead can be well-compensated by the speed of a typical disk, if the index files have to be kept on the secondary storage. In this case, our approach leads to another significant advantage: for the first time in the literature, CBR achieves the same number of direct disk accesses as FS; i.e., only one access per query term (assuming that the lists are fully read at once). Furthermore, if we assume that posting lists are kept in the main memory, which is the case for some Web search engines, the reported in-memory gains reflect overall improvements. The experimental results demonstrate the scalability and robustness of our approach.

4.6 Site-Based Dynamic Pruning for Query Processing

In the previous sections, we used incremental-CBR as a retrieval model for automatically clustered data collections. Given the efficiency improvements discussed above, we also propose to use this strategy as a dynamic pruning method for FS for the scenarios where content based clustering or categorization is not preferred or attainable due to the costs or some other limitations. That is, it is still possible to utilize incremental-CBR strategy with CS-IIS in the cases where the collection may be somehow grouped according to some basic features of the documents. One such direction can be grouping documents indexed in a search engine by their websites. This approach would be clearly much cheaper than any automatic clustering or classification approach, as well as a manual classification; yet provide efficiency gains as demonstrated above. In what follows, we first describe how our CBR strategy with CS-IIS is adapted for a site-based dynamic pruning and then present experimental results.

4.6.1 Site-Based Dynamic Pruning

In most of the commercial search engines, a typical unit of the indexing and retrieval is a single Web page[2]. That is, each page is considered as a separate entity (sometimes associated with the anchor text of the referring pages), which is indexed off-line and compared to a query on-line. On the other hand, Web pages are usually hosted by a particular organization, person, etc. and pages at the same site may form a more coherent set in terms of the content, with respect to the pages that reside in other sites. In this section, we propose to employ CBR so that first the websites that are most similar to a query are determined, and then pages within these sites and most similar to the query are returned as the final result. Our goal is to reduce the query processing time while maintaining the quality of the top-k results (where k is a small number, typically less than 30, since very few Web users look at more than the first 30 results [100]).

For a given query, we should first determine the top-S sites, namely best-sites, that are most similar to the query, and then obtain the top-k Web pages, best-pages, within these sites. Notice that, this is nothing but CBR as discussed in this thesis. That is, it can be considered as if each Web page belongs to the "cluster" identified by its website (i.e., the hostname part of its URL). Then, it is straightforward to create a CS-IIS for Web pages in the collection and use incremental-CBR strategy in this scenario. We are aware that websites may not always include semantically coherent pages, and thus, may not constitute perfect clusters. Still, we envision that for most of the sites, the overlap in terms of the content is higher for pages in a particular site than those pages that are not within this site. For instance, the findings in [101] imply that as the degree of overlap among the URLs increase, the coherency in the content (i.e., terms) in the corresponding pages also increase. This intuition is justified by the experimental results provided below.

[2]There are a few studies in the literature that discuss retrieval in coarser levels. For instance, the "logical Web document" proposed in [72] includes several actual Web pages.

4.6.2 Experiments

Dataset. In this study, we use WEBBASE collection of 4.3 million Web pages as described in Section 4.4.1. The pages in the dataset are from 1,103 websites, which constitutes the clusters in this case.

Indexing. We eliminated HTML tags, scripts, etc. and English-stop-words. No stemming is applied. Next, the typical inverted index and CS-IIS are constructed. Both files are compressed using the best performing procedures as discussed in Section 4.5.2.1. The resulting typical and cluster-skipping inverted files take 6.3 GB and 6.6 GB (uncompressed) and 767 MB and 785 MB (compressed), respectively. Note that, the increase in the CS-IIS file size is only 2%, an affordable overhead. During the experiments, only the compressed files are used.

Query processing. We use the efficiency task topics of TREC 2005 terabyte track, including 50K queries and 2.3 terms per query, on the average. The similarity computations between queries and sites/documents use *tf-idf* and the cosine metric as described before.

Effectiveness Experiments. We first compare the typical and incremental CBR strategies for site-based pruning to the baseline strategy; i.e., FS. Since there are no relevance judgments for our collection and query set, the top-k results obtained from each pruning strategy is compared to those results from the baseline. A measure based on the symmetric difference is used for comparing two lists [43]. In Figure 4.7, we plot the similarity between the top-k ($k \in \{10, 20, 30\}$) results of the baseline approach and site-based pruning approaches, namely typical and incremental strategies, versus the pruning level. The pruning level is simply controlled by the parameter S, which denotes that the top-S% of the sites is selected as the best-sites.

Our findings reveal that (i) the pruned results reveal a high similarity to the non-pruned results (e.g., incremental strategy achieves 74% similarity for top-10 results using 10% of the sites only), (ii) the incremental strategy for site-based pruning does not degrade result quality with respect to the typical strategy (as

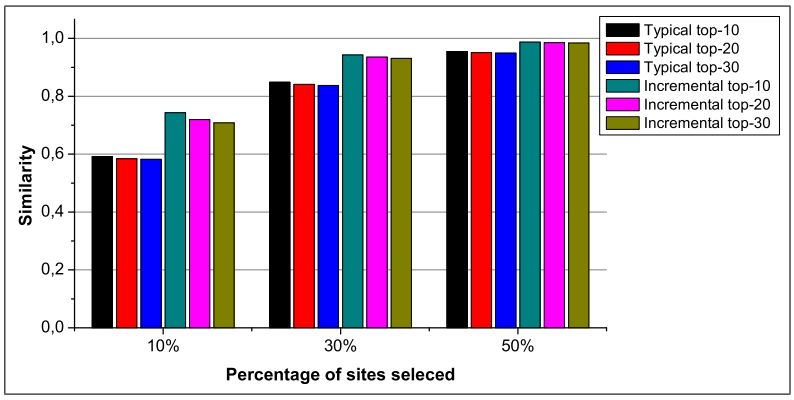


Figure 4.7: Similarity of pruned results to the baseline results.

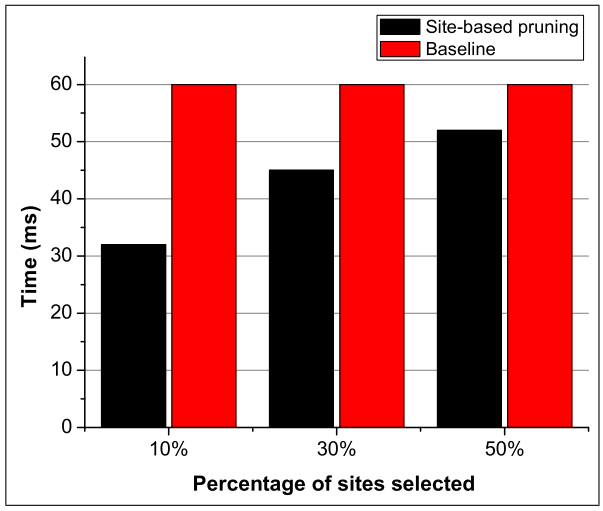


Figure 4.8: Average in-memory execution times for query processing strategies.

also observed in Section 4.5.1) and may even improve the latter, and (iii) as the number of the selected sites increase, the results converge.

Efficiency Experiments. The performance of the baseline strategy and site-based pruning strategy are compared with respect to pruning level. Note that, since the files sizes for IIS and CS-IIS are quite close for this case (in comparison to the file sizes in Section 4.5.2.1, for instance), it would be adequate to provide only in-memory execution times for baseline, namely FS, and site-based pruning approach, which is incremental-CBR with CS-IIS. Figure 4.8 reveals that the site-based pruning strategy provides significant efficiency improvements over the baseline, reaching up to 46% when the top-10% of the sites is selected.

4.6.3 Discussions

We present a dynamic query pruning technique based on incremental-CBR that eliminates relatively less promising sites (and Web pages) during retrieval. The results are encouraging in that the top-k results returned by the site-based pruning strategy exhibit strong similarity to those of the no-pruning case, while the proposed strategy achieves significant reductions in processing times.

As it is mentioned before, there are other efficient dynamic pruning techniques such as those based on the quit-continue approach [79] and impact-sorted lists [14] for FS. In Section 4.5.2.3, we have shown that incremental-CBR outperforms one of these approaches, as well. Nevertheless, it is possible for both FS and our approach to benefit from such earlier techniques. For instance, the impact-based pruning may be coupled with our site-based pruning, for further improvements in efficiency (e.g., postings for each site in CS-IIS can be sorted with respect to impacts). Exploring such possibilities is left as a future work. Another future work direction involves exploiting URL hierarchy to obtain (possibly) more coherent groups of Web pages.

4.7 Conclusions and Future Work

We introduce an incremental-CBR strategy and enhanced CS-IIS for ranking-queries. The new file organization incorporates both cluster membership and centroid information along with the usual document information into a single inverted index. In the incremental-CBR strategy, for each query term, the computations required for selecting the best-clusters and selecting the best-documents of such clusters are performed in an interleaved manner. The proposed strategy is essentially introduced for providing efficient CBR in compressed environments. We adapt multiple posting list compression parameters and a cluster-based document id reassignment technique that best fits the features of CS-IIS. We experimentally show that the proposed strategy is superior to FS for a retrieval scenario using automatically clustered datasets. Furthermore, we also show that incremental-CBR strategy can also serve as a dynamic pruning technique for FS

in a site-based pruning scenario.

The future research possibilities among others include the following. In this thesis, we concentrated on term-at-a-time query processing mode. It is also possible to use another efficient alternative, document-at-a-time processing mode, along with the proposed strategy. The proposed skip structure provides interesting data fusion [84] opportunities (i.e., merging FS and CBR results) since both of these processes can be carried out at the same time. Another interesting direction can be making the proposed system adaptive to query characteristics; during query evaluation, the number of best-clusters to be selected and the centroid term weighting schemes can be determined according to the query length or the weight distributions of the query terms. Clearly, updating our data structure is an interesting challenge. We can apply a "distributed free space" technique for future additions to posting lists. Then, given an incremental clustering algorithm (e.g., the incremental version of C^3M [35]), the complexity of updating CS-IIS is not much higher than the complexity of a typical IIS update. Yet another possible direction for improving storage and efficiency can be using skips in only "longer lists" but not in the lists of only a few words. Finally, the caching of posting lists is another topic that currently takes serious attention [18] and can be investigated in our framework, as well.

Chapter 5

Static Index Pruning with Query Views

Static index pruning techniques permanently remove a presumably redundant part of an inverted file, to reduce the file size and query processing time. In this chapter, we propose using query views in the static pruning strategies for Web search engines to improve the quality of the top-ranked results compared against the original results. The query view based strategies avoid pruning those postings that associate a term with a document, if this document has appeared among the top results of a previous query including that particular term. We incorporate query views in a number of static pruning strategies, namely term-centric, document-centric and access-based approaches, and show that the new strategies considerably outperform their counterparts especially for the higher levels of pruning and for both disjunctive and conjunctive query processing.

The rest of this chapter is organized as follows. In the next section, we provide the motivation for our research. In Section 5.2 we review the related work in the literature. In Section 5.3, we first describe the baseline pruning algorithms for this work, as discussed in [29, 43]. Next, we present an adaptive variant of the access-based pruning algorithm [58], and also propose a document-centric version. Section 5.4 introduces the new pruning strategies that exploit the query views. Section 5.5 provides an experimental evaluation of all strategies in terms of top-ranked result quality. Finally, we conclude and point to future research directions in Section 5.6.

5.1 Introduction

An inverted index is the state-of-the-art data structure for query processing in large scale information retrieval systems and Web search engines (WSEs) [122]. In the last decades, several optimizations have been proposed to store and access inverted index files efficiently, while keeping the quality of the search relatively stable (see Chapter 3.2.1). One particular method is static index pruning, which aims to reduce the storage space and query execution time.

The sole purpose of a static pruning strategy is staying loyal to the original ranking of the underlying search system for most queries, while reducing the index size, to the greatest extent possible. This is a non-trivial task, as it would be impossible to generate exactly the same results as produced by an unpruned index for all possible queries. Most pruning strategies attempt to provide quality guarantees for only top-ranked results, and try to keep in the pruned index those terms or documents that are the most important according to some measure, hoping that they would contribute to the future query outputs uttermost. The heuristics and measures used for deciding which items should be kept in the index and which of them should be pruned distinguish the static pruning strategies. Many proposals in the literature are solely based on the features of the collection and search system. For instance, in one of the pioneering works, Carmel et al. sort the postings in each term's list with respect to the search system's scoring function and remove those postings with the scores under a threshold [43]. This is said to be a term-centric approach. In an alternative document-centric strategy, instead of considering posting lists, pruning is carried out for each document [29]. These two strategies, as well as some others reviewed in the next section essentially take into account the collection-wide features (such as term frequency) and search system features (such as scoring functions).

However, in the case of Web search, additional sources of information are also available that may enhance the pruning process and final result quality, which is the most crucial issue for search engines. In this sense, query logs serve as an invaluable source of information: in the world of (theoretically) infinitely many combinations of possible query terms, the query logs highlight those terms and

combinations that are important enough to be searched in the past. Thus, these logs can provide further insight and evidence on which terms or documents should be kept in a pruned index to answer the future queries.

In a recent pruning strategy that explicitly makes use of the previous query logs [58, 59, 60] the notion of access frequency is employed. That is, the pruning strategy is guided by the number of appearances of a document in the query outputs. In this work, we propose a new pruning heuristic that exploits query views. That is, the pruning process is also guided by considering the actual query terms that access to the documents.

In the literature, the idea of using query terms to represent a document is known as query view [44]. In the scope of our work, all queries that rank a particular document among their top-ranked results constitute the query view of that document. For static pruning purposes, we exploit the query views in the following sense. We envision that, for a given document d and a term t in d, the appearance of t in d's query view is the major evidence of its importance for d; i.e., it implies that t is a preferred way of accessing document d in the search system. Thus, any pruning strategy should avoid pruning the index entry d from the posting list of term t to the greatest extent possible.

In this work, our goal is improving the quality of the results obtained from a pruned index, which has vital importance for the WSEs in a competitive market. To this end, we introduce new pruning approaches that incorporate the query view idea into the term-centric [43], document-centric [29] and access-based [58] strategies in the literature. We show that, the pruning strategies with the query view significantly improve the quality of the top-ranked results, especially at the higher levels of pruning. More concretely, our contributions in this chapter are as follows:

- First, we fully explore the potential of a previous strategy, namely access-based pruning, that also makes use of the query logs in the static index pruning context. To this end, we provide an adaptive version of the term-centric

pruning algorithm provided in [58]. We also introduce a new document-centric version of the access-based algorithm, and show that the latter outperforms its term-centric counterpart.

- Second, we provide an effectiveness comparison of these access-based approaches to the term-centric approach [43] and document-centric approach [29], for their best performing setups reported in the literature. Our experimental findings reveal that, although the access based methods are inferior to the latter strategies for disjunctive query processing (as shown in the literature [58]), they turn out to be the most effective strategies when the queries are processed in the conjunctive mode. This is a new result that has not been reported before. Furthermore, the document-centric version of the access-based strategy as described here is found to be superior to all other strategies for conjunctive query processing, which has utmost importance for WSEs.

- Finally, the main contribution of this chapter is exploiting query views to tailor more effective static index pruning strategies for both disjunctive and conjunctive query processing; i.e., the most common query processing modes in WSEs [55]. More specifically, the terms of a document that appear in the query view of this particular document are considered to be privileged and preserved in the index to the greatest possible extent during the static pruning. The query view heuristic is coupled with all three pruning approaches in the literature (term- and document-centric approaches as proposed in [29, 43], and the access-based term-centric method adapted from [58]) as well as the document-centric version of the access-based method that is introduced here.

Our findings reveal that for both disjunctive and conjunctive query processing, the query view based pruning strategies reveal an excellent performance in terms of the similarity of the top-ranked results to the original results (i.e., those obtained by using the original index) and significantly outperform their counterparts without query views. The gains are especially emphasized at the higher levels of pruning. We also verify our findings using training logs of varying number

of queries and a very large test set including 100,000 queries.

Furthermore, the improvements provided by the query view based strategies also apply to the cases where the pruned index is not used to replace the original index, but rather used as a list cache (as in the ResIn framework [104]) for efficiency purposes. In the latter setup, the essential requirement for a pruning strategy is being able to provide correctness guarantee (i.e., producing exactly the same results as the main index) for the highest number of queries. We show that our query view based pruning strategies can output the correct result for a considerably more number of queries than the baseline algorithms; i.e., those without query views. This means that pruned index files that are created using the query view based strategies can either replace the original index, say, at the back-end servers, or serve as a front-end cache in WSEs.

5.2 Related Work

5.2.1 Static Inverted Index Pruning

In the last decade, a number of different approaches have been proposed for the static index pruning. In this study, as in [29], we use the expressions *term-centric* and *document-centric* to indicate whether the pruning process iterates over the terms (or, equivalently, the posting lists) or the documents at the first place, respectively. Note that, this terminology is slightly different than that of [43]. Additionally, we call a strategy *adaptive* if its pruning criteria (e.g., a threshold) dynamically changes for different terms or documents. In contrast, a *uniform* strategy applies pruning with a fixed threshold for all documents or terms.

In one of the earliest works in this field, Carmel et al. proposed term-centric approaches with uniform and adaptive versions [43]. In this work, an idealized top-k pruning algorithm is introduced, which is guaranteed to generate the same answers (within an error of ϵ) as the original index for queries including less than $1/\epsilon$ terms. It is observed that this idealized algorithm provides only negligible pruning effects, and thus it is relaxed by a score-shifting operation. After this

latter modification, which also relaxes the theoretical guarantees, the adaptive version of the algorithm is reported to provide substantial pruning of the index and exhibit excellent performance at keeping the top-ranked results intact in comparison to the original index. Roughly, adaptive top-k algorithm sorts the posting list of each term according to some scoring function (e.g., Smart's *tf-idf* in [43]) and removes those postings that have scores under a threshold determined for that particular term. In our study, this algorithm (which is referred to as *TCP strategy* hereafter) is employed as a baseline pruning strategy and its further details are discussed in Section 5.3.1.

In [55], the authors propose an index pruning approach that is tailored to support conjunctive and phrase queries, which requires a positional index. In this strategy, the term co-occurrence information is used to guide the pruning. In a nutshell, this strategy has three stages. First, the most significant sentences of the documents are selected. Next, these sentences are ranked and a fixed number of them are selected. Finally, the frequency and positional index files are constructed so that they only consider those terms and their positional information that appear in the selected sentences. In a follow-up work, a more sophisticated algorithm with the same goals is proposed [54].

In [22], another term-centric pruning strategy is suggested. In this work, the collection dependent stop-words are identified and totally removed from the index. To determine those terms to be pruned, several measures like inverse document frequency (idf), residual idf and term discriminative value are used. Their findings indicate that, although this approach can outperform the TCP strategy for some cases, the latter is better for short queries and obtaining high P@10 scores. This justifies our choice of TCP to be used in this work, as we essentially focus on improving the result quality for Web queries over a pruned index.

Another recently proposed term-centric pruning approach is based on the probability ranking principle [20]. Briefly, for each document in a term's posting list, this strategy computes a score that represents the significance of that term to the document, and prunes those that are below a global threshold. This approach

is shown to be superior to TCP in terms of MAP results; however its performance for P@10 is less stable, but still comparable with TCP.

Finally, the access-based static pruning strategy discussed in [58] employs a query log and computes the number of appearances of each document in top-1000 results of the queries. These access-counts are then used to guide the pruning of posting lists for each term in the lexicon; i.e., in a term-centric fashion. This strategy is uniform, in the sense that for each term, a fixed number of postings that belong to the documents with highest access-count scores are stored in the pruned index, and the rest is pruned. In [58], the performance of this algorithm is shown to be somewhat discouraging, and as a remedy, the authors devise a mechanism to predict the query difficulty. Then, "simple" queries are processed by the pruned index, whereas "difficult" ones are forwarded to the original index, which should also be stored. In this study, we provide an adaptive version of the term-centric approach outlined above. We also propose a document-centric version, which outperforms the former one. Further details of this approach are discussed in Section 5.3.2.

Note that, the access-based pruning approach is also adapted for dynamic pruning [59, 60]. In that case, the query processing dynamically stops when a threshold is reached while processing a query term's posting list, which is sorted in access-count order. This approach is out of the scope of our thesis and not elaborated further.

As an alternative to term-centric pruning, Büttcher et al. proposed a document-centric pruning (referred to as *DCP* hereafter) approach with uniform and adaptive versions [29]. In the DCP approach, only the most important terms are left in a document, and the rest are discarded. The importance of a term for a document is determined by its contribution to the document's Kullback-Leibler divergence (KLD) from the entire collection. However, the experimental setup in this latter work is significantly different than that of [43]. That is, only the most frequent terms of the collection are pruned and the resulting (relatively small) index is kept in the memory, whereas the remaining unpruned body of index resides on the disk. During retrieval, if the query term is not found in

the pruned index in memory, the unpruned index is consulted. In a more recent study [9], a comparison of TCP and DCP for pruning the entire index is provided in a uniform framework. It is reported that for disjunctive query processing TCP essentially outperforms DCP for various parameter selections. In this work, we also use the DCP strategy to prune the entire index, and employ it as one of the baseline strategies (see Section 5.3.1).

In most of the above works, it is either explicitly or implicitly assumed that the pruned index will replace the original one (e.g., at the back-end servers in a WSE), and the pruning strategies are optimized for providing the most similar results to the original result. In this sense, these pruning approaches can be considered as lossy. In another line of research, it is proposed to use a pruned index only for efficiency purposes while also keeping the original index in the system, so that the correctness of the queries can be always guaranteed. To this end, Ntoulas and Cho describe pruning strategies with correctness guarantees [83]. A similar approach is also taken in the ResIn framework [104]. In ResIn, it is assumed that a pruned index is placed between the WSE front-end and the broker, which is responsible for sending the queries to the back-end servers with the main index. In this case, the pruned index serves as a posting list cache, and the queries are passed to the broker and the back-end only when it is deduced that the query cannot be answered correctly. The originality of ResIn lies in its realistic architecture that also takes into account a dynamic result cache placed in front of the pruned index and the back-end. That is, all queries are filtered through the result cache, and only the misses are sent to the pruned index and/or back-end servers. Thus, the pruning algorithms employed in such an architecture should perform well essentially for the miss-queries. Their experiments show that keeping the full posting lists for the most popular query terms in the pruned index serves well for the miss queries, whereas pruning lists (as in TCP and DCP) performs worse. A combination of both techniques is shown to provide substantial increase in the hit rates, or equivalently, in the number of queries that can be answered correctly with the pruned index.

In this work, we consider both possible usages of a pruned index: either at the back-end servers in a lossy manner, or at the front-end as a list-cache. In

Section 5.5, we report results for both usages; i.e., in terms of similarity to the original results and the number of queries that are answered correctly by each pruning strategy. These experiments reveal that our query view based strategies provide significant improvements in the result quality and yield pruned index files that can be utilized in both scenarios.

5.2.2 Query Views for Representing Documents

Query logs are exploited in several ways in the information retrieval literature. In the scope of this work, we only focus on the related work for their usage as a representation model for documents. The concept of "query view" is first defined in [44]. In this work, queries are used as features for modeling documents in a web site. [92] also uses queries for document representation (called "query vector model") in the context of document selection algorithms for parallel information retrieval systems. In this work, each query is associated with its top-k resulting documents and no click information is used. This is similar to our case, as we also restrict the notion of the query view only to the output of the underlying search engine and disregard the click-through information. This choice makes sense for the purposes of pruning, as the aim of a static pruning algorithm is generating the same or most similar output with the underlying search system.

In a recent work [91], query log is mined to find "frequent query patterns", which form the "query-set model". Then each document is represented by the query-set model for clustering documents in a web site. This work suggests that query based representation dramatically improves the quality of the results. Another recent work [16] uses query terms as tags to label the documents that appear in the top-k results and are clicked by the users.

5.3 Static Pruning Approaches

We start with describing how exactly TCP and DCP algorithms are implemented in our framework. Next, we describe access-based TCP, as a slightly modified version of Garcia's uniform pruning algorithm [58]. Finally, we introduce

Algorithm 6 Term-Centric Pruning (TCP)

Input: I, k, ϵ, N

```
1: for each term t ∈ I do
2:    fetch I_t from I
3:    if |I_t| > N/2 then
4:       remove I_t entirely from I
5:    if |I_t| > k then
6:       for each posting (d, f_{d,t}) ∈ I_t do
7:          compute Score(t, d) with BM25
8:       z_t ← k^{th} highest score among the scores
9:       τ_t ← z_t × ε
10:      for each posting (d, f_{d,t}) ∈ I_t do
11:         if Score(t, d) ≤ τ then
12:            remove entry (d, f_{d,t}) from I_t
```

a document-centric version of the latter strategy.

5.3.1 Baseline Static Pruning Algorithms

Term-Centric Pruning (TCP) strategy. As it is mentioned in the previous section, TCP, the adaptive version of the top-k algorithm proposed in [43], is reported to be very successful in static pruning. In this strategy, for each term t in the index I, first the postings in t's posting list are sorted by a scoring function (e.g, *tf-idf*). Next, the k^{th} highest score, z_t, is determined and all postings that have scores less than $z_t \times \epsilon$ are removed, where ϵ is a user defined parameter to govern the pruning level. Following the practice in [21], we disregard any theoretical guarantees and determine ϵ values according to the desired pruning level.

In a recent study, it is shown that the performance of the TCP strategy can be further boosted by carefully selecting and tuning the scoring function used in the pruning stage [21]. Following the recommendations of that work, we employ BM25 as the scoring function for TCP and entirely discard the terms with document frequency $f_t > N/2$ (where N is the total number of documents) as their BM25 score turns out to be negative. In Algorithm 6, we demonstrate TCP strategy as adapted in our framework.

Algorithm 7 Document-Centric Pruning (DCP)

Input: D, λ
1: **for** each document $d \in D$ **do**
2: sort $t \in d$ in descending order w.r.t. $Score(d, t)$
3: remove the last $|d| \times \lambda$ terms from d

Document-Centric Pruning (DCP) strategy. In this work, we apply the DCP strategy for the entire index, which is slightly different than pruning only the most frequent terms as originally proposed by [29]. Additionally, instead of scoring each term of a document with KLD, we prefer to use BM25, to be compatible with TCP. In a recent work, BM25 is reported to perform better than KLD for DCP, as well [9]. Finally, in [29] it is again shown that the uniform strategy; i.e., pruning a fixed number of terms from each document, is inferior to the adaptive strategy, where a fraction (λ) of the total number of unique terms in a document is pruned. Algorithm 7 conveys the DCP strategy.

5.3.2 Adaptive Access-based Static Pruning Strategies

Access-based Term-Centric Pruning (aTCP) strategy. For the first time in the literature, Garcia et al. used the search engine query logs to guide the static index pruning process [58]. However, their work does not use the actual content of the queries, but just makes use of the access count of a document; i.e., the number of times a document appears in top-k results of queries, where k is typically set to 1000. Furthermore, they essentially focus on the dynamic index pruning [58, 59, 60], and propose a rather simple algorithm for the static case. In particular, their algorithm applies the, so-called, MAXPOST heuristic, which simply keeps a fixed number of postings with the highest number of access count in each term's posting list.

The result of the MAXPOST approach is not very encouraging. Despite considerable gains (up to 75%) in the query processing time, the reductions in accuracy is significant; i.e., up to 22% drop in MAP is observed when only 35% of the index is pruned (see [58, p. 114, Figure 5.2]). We attribute this result to the uniform pruning heuristic, which is shown to be a relatively unsuccessful

Algorithm 8 Access-based Term-Centric Pruning(aTCP)

Input: I, μ, AccessScore[]
1: **for** each term $t \in I$ **do**
2: fetch I_t from I
3: sort $(d, f_{d,t}) \in I_t$ in descending order w.r.t. AccessScore[d]
4: remove the last $|I_t| \times \mu$ postings from I_t

Algorithm 9 Access-based Document-Centric Pruning (aDCP)

Input: D, μ, AccessScore[]
1: sort $d \in D$ in descending order w.r.t. AccessScore[d]
2: $numPrunedPostings \leftarrow 0$
3: **while** $numPrunedPostings < |D| \times \mu$ **do**
4: remove the document d with the smallest access score
5: $numPrunedPostings \leftarrow numPrunedPostings + |d|$

approach for other strategies (e.g., TCP and DCP) as discussed above.

For this study, we decide to implement an adaptive version of the MAXPOST approach. Since it iterates over each term and removes some postings, we classify this approach as term-centric, and call the adaptive version *access-based TCP* (aTCP). In this case, instead of keeping a fixed number of postings in each list, we keep a fraction (μ) of the number of postings in each list. Algorithm 8 shows aTCP strategy.

Access-based Document-Centric Pruning (aDCP) strategy. In this thesis, we propose a new access-based strategy. Instead of pruning the postings from each list, we propose to prune documents entirely from the collection, starting from the documents with the smallest access counts. The algorithm is adaptive in that, for an input pruning fraction (μ), the pruning iterates while the total length of pruned documents is less than $|D| \times \mu$, where $|D|$ is the collection length; i.e., total number of unique terms in the collection. Algorithm 9 presents this strategy, which we call *access-based DCP* (aDCP).

Note that, for both of the access-based approaches (aTCP and aDCP) many documents may have the same access count. To break the ties, we need a secondary key to sort these documents. In this study, we simply use the URL of the

Web pages and sort those documents with the same access count in lexicographical order. It is also possible to consider the length of the document, or the length of its URL, which are left as a future work.

5.4 Static Index Pruning Using Query Views

In this section, we first define the notion of query view (QV) for a document, and then introduce the pruning strategies that incorporate the query view heuristic. Let us assume a document collection $D = \{d_1, \cdots, d_N\}$ and a query log $Q = \{Q_1, \cdots, Q_M\}$, where $Q_i = \{t_1, \cdots, t_q\}$. After this query log Q is executed over D, the top-k documents (at most) are retrieved for each query Q_i, which is denoted as $R_{Q_i,k}$. Now, we define the query view of a document d as follows:

$QV_d = \cup Q_i$, where $d \in R_{Q_i,k}$

That is, each document is associated with a set of terms that appear in the queries which have retrieved this document within the top- k results. Without loss of generality, we assume that during the construction of the query views, queries in the log are executed in the conjunctive mode; i.e., all terms that appear in the query view of a document also appear in the document.

The set of query views for all documents, QV_D, can be efficiently computed either offline or online. In an offline computation mode, the search engine can execute a relatively small number of queries on the collection and retrieve, say, top-1000 results per query. Note that, as discussed in [59], it may not be necessary to use all of the previous log files; the most recent log and/or sampling from the earlier logs can be sufficiently representative. In Section 5.5, we show that even small query logs (e.g., of 10K queries with top-1000 results) provide gains in terms of effectiveness. On the other hand, in the online mode, each time a query response is computed, say, top-10 results (i.e., only document ids) for this query can also be stored in the broker (or, sent to a dedicated query view server). Note that, such a query view server can store results for millions of queries in its secondary storage to be used during the index pruning, which is actually an offline

Algorithm 10 Term-Centric Pruning with Query Views (TCP-QV)

Input: I, k, ϵ, N, QV_D

```
 1: for each term t ∈ I do
 2:     fetch I_t from I
 3:     if |I_t| > N/2 then
 4:         remove I_t entirely from I
 5:     if |I_t| > k then
 6:         for each posting (d, f_{d,t}) ∈ I_t do
 7:             compute Score(t, d) with BM25
 8:         z_t ← k^t h highest score among the scores
 9:         τ_t ← z_t × ε
10:         for each posting (d, f_{d,t}) ∈ I_t do
11:             if (Score(t, d) ≤ τ and t ∉ QV_d then
12:                 remove entry (d, f_{d,t}) from I_t
```

process. In the experiments, we also provide the effectiveness figures obtained for the query views that are created by using only top-10 results.

We exploit the notion of query views for static index pruning, as follows. We envision that for a given document, the terms that appear as query terms to rank this document within top results of these queries should be privileged, and should not be pruned to the greatest extent possible. That is, as long as the target pruned index size is larger than the total query view size, all query view entries are kept in the index. In what follows, we introduce four pruning strategies that exploit the query views, based on the TCP, DCP, aTCP and aDCP strategies, respectively.

Term-Centric Pruning with Query Views (TCP-QV). This strategy is based on Algorithm 6, but employs query views during pruning. In particular, once the pruning threshold (τ_t) is determined for a term t's posting list, the postings that have scores below the threshold are not directly pruned. That is, given a posting d in the list of term t, if $t \in QV_d$, this posting is preserved in the index, regardless of its score. This modification is presented in Algorithm 10. Note that, by only modifying line 11, the query view heuristic is taken into account to guide the pruning.

Document-Centric Pruning with Query Views (DCP-QV). In this case,

Algorithm 11 Document-Centric Pruning with Query Views (DCP-QV)

Input: D, λ, QV_D

```
1: for each document d ∈ D do
2:     for each term t ∈ d do
3:         if t ∈ QV_d then
4:             Pr_t ← 1
5:         else
6:             Pr_t ← 0
7:     sort t ∈ d in descending order w.r.t. first Pr_t then Score(d, t)
8:     remove the last |d| × λ terms from d
```

Algorithm 12 Access-based Term-Centric Pruning with Query Views (aTCP-QV)

Input: I, μ, AccessScore[], QV_D

```
1: for each term t ∈ I do
2:     fetch I_t from I
3:     for each posting (d, f_{d,t}) ∈ I_t do
4:         if t ∈ QV_d then
5:             Pr_d ← 1
6:         else
7:             Pr_d ← 0
8:     sort (d, f_{d,t}) ∈ I_t in descending order w.r.t. first Pr_d then AccessScore[d]
9:     remove the last |I_t| × μ postings from I_t
```

for the purpose of discussion, let us assume that each term t in a document d is associated with a priority score Pr_t, which is set to 1 if $t \in QV_d$ and 0 otherwise. The terms of a document d are now sorted (in descending order) according to these two keys, first the priority score and then score function output. During the pruning, last $|d| \times \lambda$ terms are removed, as before. This strategy is demonstrated in Algorithm 11.

Access-based Term-Centric Pruning with Query Views (aTCP-QV). In aTCP strategy, again for the purposes of discussion, we assume that each posting d in the list of a term t is associated with a priority score Pr_d, which is set to 1 if $t \in QV_d$ and 0 otherwise. Then, the postings in the list are sorted in the descending order of the two keys, first the priority score and then the access count. During the pruning, last $|I_t| \times \mu$ postings are removed (Algorithm 12).

Algorithm 13 Access-based Document-Centric Pruning with Query Views (aDCP-QV)

Input: D, μ, AccessScore[], QV_D
1: sort $d \in D$ in descending order w.r.t. AccessScore[d]
2: $numPrunedPostings \leftarrow 0$
3: **while** $numPrunedPostings < |D| \times \mu$ **do**
4: fetch d with the smallest score
5: **for** each term $t \in d$ **do**
6: **if** $t \notin QV_d$ **then**
7: remove t from d
8: $numPrunedPostings \leftarrow numPrunedPostings + 1$

Access-based Document-Centric Pruning with Query Views (aDCP-QV). In this case, we again prune the documents starting from those with the smallest access counts until the pruning threshold μ is reached. But, while pruning documents, those terms that appear in the query view of these documents are kept in the index. This is shown in Algorithm 13. Note that, for a given pruning threshold, this algorithm would possibly prune documents with higher access counts than its counterpart without query views (aDCP).

Note that, in Algorithms 10, 11, 12 and 13, we show the use of query views in a simplistic manner for the purposes of discussion, without considering the actual implementation. For instance, for TCP-QV case, it would be more efficient to first create an inverted index of the QV_D and then process the original index and query view index together; i.e., in a merge-join fashion, for each term in the vocabulary. We presume that for all four approaches employing query views, the additional cost of accessing an auxiliary data structure for QV_D (either the actual or inverted data) would be reasonable, given that the query terms highly overlap and only a fraction of documents in the collection have high access frequency [60]. Furthermore, it is not necessary to use all previous query logs, as discussed above [59]. Therefore, we expect that the size of the data structures for query views would be much smaller when compared to the actual collection, i.e., Web.

5.5 Experimental Evaluation

5.5.1 Experimental Setup

Document collection and indexing. For this study, we obtained the list of URLs that are categorized at the Open Directory Project (ODP) Web directory [85]. Among these links, we successfully crawled around 2.2 million pages, which take 37 GBs of disk space in uncompressed HTML format. This constitutes our document collection for this study.

We first indexed the dataset using the publicly available Zettair IR system [121]. During the indexing, Zettair is executed with the "no stemming" option. All stop-words and numbers are included in the index, yielding a vocabulary of around 20 million unique terms. Once the initial index is generated, we used our homemade IR system to create the pruned index files and execute the training and test queries over them.

Query log normalization. We use a subset of the AOL Query Log[1] that contains 20 million queries of about 650K people for a period of 12 weeks. The query terms are normalized by case-folding, sorting in the alphabetical order and removing the punctuation and stop-words. We consider only those queries of which all terms appear in the vocabulary of the collection. This restriction is forced to guarantee that the selected queries are more sensible for the dataset.

Training and test query sets. From the normalized query log subset, we construct training and test sets. The training query sets that are used to compute the access counts and query views for the documents are from the first half (i.e., 6 weeks) of the log. The test sets that are used to evaluate the performance for different pruning strategies are constructed from the second half (last 6 weeks) of the log. During the query processing with both training and test sets, a version of BM25 scoring function, as described in [29], is used.

In the training stage, queries are executed in the conjunctive mode and top-k

[1] `http://imdc.datcat.org/collection/1-003M-5`

Table 5.1: Characteristics of the training query sets

	10K-top1000	50K-top1000	518K-top1000	1.8M-top1000	518K-top10	1.8M-top10
Access %	30%	54%	79%	85%	33%	50%
QV Size (%)	35MB(1%)	143MB(4%)	647MB(20%)	1,093MB(34%)	53MB(2%)	148MB(5%)

results per query are retrieved to compute the access counts and query views. To observe the impact of the training set size, we created training sets of 10K, 50K, 518K and 1.8M distinct queries that are selected randomly from the first half of the log and obtained top-1000 results per query. To further investigate the impact of the result set size, namely, k, we obtained only top-10 results for the latter two training sets (i.e., including 518K and 1.8M queries). Thus, we have six different training query logs with varying number of queries and results per query. Characteristics of the training sets are provided in Table 5.1.

In the first row of Table 5.1, we provide the access percentage achieved by each training set; i.e., the percentage of documents that appear at least once in a query result. In the second row of the table, we report the percentage of the total query view size to the collection size, where the former is the sum of the number of unique query terms that access to a document and the latter is the sum of the number of unique terms per document, as usual. Both values increase as the number of queries increase, however the increments follow a sub-linear trend. This is due to the heavy-tailed distribution of accesses to documents as shown before [58].

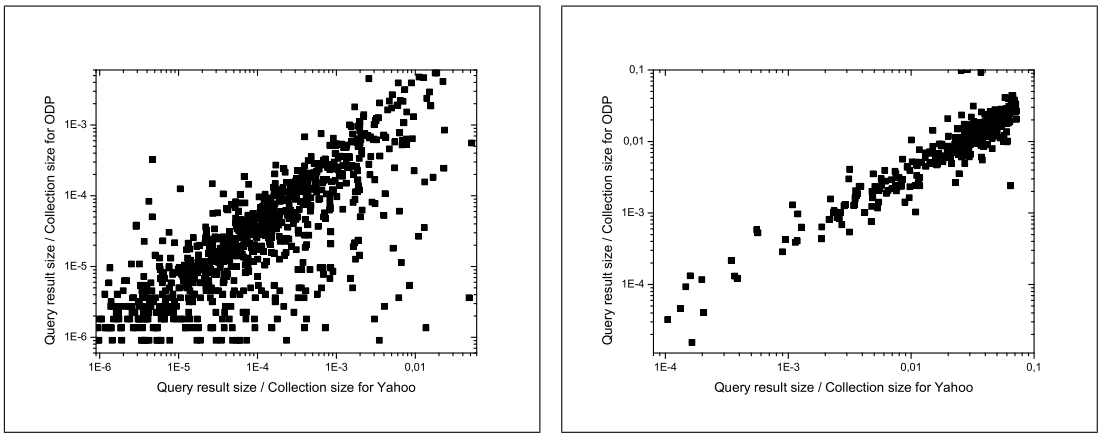
Remarkably, access percentages for 10K-top1000 and 518K-top10 training sets are very close, which imply that access counts and query views with similar characteristics can be either obtained by using a relatively small query log and larger number of results, or using a larger query log but retrieving smaller number of, say only top-10, results. The former option can be preferred during an offline computation, whereas the latter can be achieved for an online computation. For instance, a search engine can store the top-10 document identifiers per query (maybe at a dedicated server) on the fly to easily compute the query views when required. Note that, these observations are also valid for the 50k-top1000 vs. 1.8M-top10 sets. In the experiments, we show that these sets also yield relatively similar effectiveness figures.

For the majority of the experiments reported in the next section, we use a test set of 1000 randomly selected queries from the second half of the AOL log. These queries are normalized as discussed above. We keep only those queries that can retrieve at least one document from our collection when processed in the conjunctive mode. By definition, the test set is temporally disjoint from the training sets. Furthermore, we guarantee that train and test sets are query-wise disjoint by removing all queries from the test set that also appear in the training sets (after the normalization stage). But, some of the terms in the queries in both sets, of course, may overlap. This set is referred to as test-1000 in the following sections.

Note that, this latter elimination of overlapping queries with the training sets yields a test set of queries that are in the heavy tail of the query log. That is, we are left with the queries that appear (almost) only once in the test set (as more frequent queries also occur in the training sets and thus eliminated). In this sense, our test set is similar to the "miss-queries" as described by the ResIn architecture [104]; i.e., those queries that cannot be found in the result-cache and forwarded to the pruned index. In our case, removal of queries that also occur in the training set results in a test set of singleton queries, which cannot be cached neither dynamically nor statically. Thus, our performance improvements/findings obtained on this test set would be valid for both possible usages of a pruned index, either as a front-end cache (as in [83, 104]) or at the back-end servers, in a Web search engine.

In what follows, we conducted experiments for both disjunctive and conjunctive processing of the queries using the test-1000 set. For each case, top-1000 results are retrieved for evaluation purposes.

Compatibility of the dataset and query sets. As discussed in [115], the compatibility of the query log and underlying document collection is a crucial issue for the reliability of an experimental framework. Intuitively, we consider that our dataset and query log are compatible, since the ODP site is a general Web directory consisting of pages from several different categories, and AOL log is a general search engine log. To experimentally justify this claim, we further



(a) (b)

Figure 5.1: The correlation of "query result size/collection size" on ODP and Yahoo for: (a) conjunctive, and (b) disjunctive query processing modes.

conducted a preliminary experiment as follows. We processed the test-1000 set both in conjunctive and disjunctive modes on our collection, and recorded the total number of results per query. Next, we also submitted the same queries to a major search engine, Yahoo! (using its Web API) again in conjunctive (default) and disjunctive processing modes. For each case, we also stored the number of results per query as returned by the search engine API. We assume that the underlying collection of Yahoo! includes around 31.5 billion pages, which is the reported number of results when searching for the term "a" at Yahoo! Web site.

For conjunctive query processing, ODP and Yahoo! collections yield 398 and 29,907,586 results on the average, which corresponds to 1.78×10^{-4} and 9.5×10^{-4} of the underlying collection size, respectively. For disjunctive processing, ODP and Yahoo! produces 25K and 857 million results on the average, again corresponding to 0.01 and 0.03 of the searched collections, respectively. In Figure 5.1, we represent the test-1000 queries on a log-log scale plot where the y-axis is the ratio of the number of results retrieved in our ODP collection to the collection size, and the x-axis is the same ratio for Yahoo! collection, for conjunctive and

disjunctive[2] query processing. The figure also reveals that the ratio of the results per query in each collection are positively correlated, i.e.; yielding correlation coefficients of 0.59 and 0.67 for conjunctive and disjunctive modes. Thus, we conclude that our collection and query sets are compatible and the experimental evaluations would provide meaningful results.

Evaluation measure. In this work, we compare the top-k results obtained from the original index against the pruned index, where k is 10 (the results for k =2, 100 and 1000 reveal similar trends and are not reported here to save space). To this end, we employ the symmetric difference measure as discussed in [43]. That is, for two top-k lists, if the size of their union is y and the size of their symmetric difference is x, symmetric difference score $s = 1 - x/y$. The score of 1 means exact overlap, whereas the score of 0 implies that two lists are disjoint. The average symmetric difference score is computed over the individual scores of 1,000 test queries and reported in the following experiments. Note that, symmetric difference measure does not take into account the order of the results. To this end, it is possible to use a measure based on Kendall's tau [43], which is left as a future work.

Parameters for the pruning strategies. The pruned index files are obtained at the pruning levels ranging from 10% to 70% (with a step value of 10%) by tuning the ϵ, λ and μ parameters in the pruning algorithms. All index sizes are considered in terms of their raw (uncompressed) sizes. For TCP, top-k parameter is set to 10 during pruning. Our preliminary experiments revealed that for all strategies, updating the document lengths after the index pruning stage does not provide any gains, and thus original document lengths are used by BM25 during the query processing.

[2]Yahoo! Web API reports a fixed number of results, 2^{31}, for queries that produce more results than that number, which especially occurs in the disjunctive mode. For this case, we exclude these queries and report the correlation values for the remaining 373 queries on both collections.

Table 5.2: Average symmetric difference scores for top-10 results and disjunctive query processing (relative improvements with respect to the baseline algorithm are shown in the column Δ%; all improvements are statistically significant)

%	TCP	DCP	aTCP	aDCP	TCP-QV	Δ%	DCP-QV	Δ%	aTCP-QV	Δ%	aDCP-QV	Δ%
10%	0.97	0.94	0.84	0.94	0.98	1%	0.98	4%	0.93	11%	0.96	2%
20%	0.91	0.86	0.68	0.87	0.95	4%	0.95	10%	0.88	29%	0.91	5%
30%	0.83	0.77	0.54	0.77	0.91	10%	0.93	21%	0.84	56%	0.86	12%
40%	0.74	0.68	0.42	0.66	0.86	16%	0.89	31%	0.80	90%	0.81	23%
50%	0.64	0.58	0.31	0.54	0.82	28%	0.84	45%	0.76	145%	0.77	43%
60%	0.55	0.49	0.22	0.41	0.79	44%	0.79	61%	0.74	236%	0.74	80%
70%	0.47	0.40	0.14	0.30	0.71	51%	0.66	65%	0.62	343%	0.66	120%

5.5.2 Results

Statistical significance of the results. All results reported in the below sections, unless stated otherwise, are found to be statistically significant at 0.05 level. In particular, for the results in Tables 5.2 and 5.3, and Figures 5.2 and 5.3, at each pruning level, the output of 1000 test queries for a baseline algorithm and its query view based counterpart are compared using the paired t-test and Wilcoxon signed rank test. For Tables 5.2 and 5.3, there are only two cases where the query views do not improve the performance and, subsequently, there is no statistical difference in means. For the cases in Figures 5.2 and 5.3, there are only a few cases where a query view based strategy yields no significant improvements (especially for smaller training sets) and these cases are discussed later. Additionally, for the results shown in Tables 5.2 and 5.3, we made a one-way ANOVA analysis (followed by Tukey's post hoc test) among the four baseline strategies as we also compare their performance in the following sections. It is found that only in Table 5.2, the mean scores are not significantly different at 0.05 level between DCP and aDCP up to 40% pruning level. All other means are pairwise different according to Tukey test results.

Performance of the query views: disjunctive mode. In Table 5.2, we provide average symmetric difference results of all eight pruning strategies for the top-10 results and disjunctive query processing mode. For access-based and query view based strategies, we employed our largest training set, namely, 1.8M-top1000. In terms of the four baseline algorithms, the findings in this case confirm the earlier observations in [9, 43, 58]. Our adaptation of the access-based

approach, aTCP, is the worst among all and only after 30% pruning, the symmetric difference score drops down to 0.54. On the other hand, the document-centric version of the access-based pruning strategy, aDCP, achieves much better performance; it is clearly superior to its term-centric counterpart and provides comparable results to DCP, at the early stages of the pruning (up to 50%). Among these four strategies, TCP is the clear winner whereas DCP is the runner-up and the access-based strategies are inferior to those, especially at the higher levels of pruning. This implies that solely using access counts is not adequate to guide the static index pruning.

Next, we evaluate the performance of the strategies with query views, namely TCP-QV, DCP-QV, aDCP-QV and aTCP-QV. A brief glance over Table 5.2 reveals that these approaches are far superior to their counterparts that are not augmented with query views. Remarkably, the order of algorithms is similar in that TCP-QV is still the best performer (though sometimes replaced by DCP-QV) and aTCP-QV is the worst. However, the gaps are now considerably closer. Indeed, the percentage improvement columns reveal that, query views enormously enhance the performance of the poor strategies (e.g., aTCP) at all pruning levels (ranging from 11% to 343%). Even for those strategies that were relatively more successful before, query views provide significant gains, especially at the higher levels of the pruning. For instance, at 50% pruning, the symmetric difference score jumps from 0.64 to 0.82 for TCP (a relative increase of 28%), and from 0.58 to 0.84 for DCP (45%). The relative improvements for all strategies exceed 10% after 20% pruning level. In short, query views significantly improve the baseline strategies, and carry them around 75-80% effectiveness at 40-50% pruning level, which is a solid success.

Performance of the query views: conjunctive mode. In Table 5.3, we provide symmetric difference results in the same setup but for conjunctive query processing mode. Interestingly, conjunctive processing is mostly overlooked and has been taken into account in only few works [54, 55, 104], whereas it is the default and probably the most crucial processing mode for WSEs. Thus, we first analyse the results for the baseline strategies, which has not been discussed in the literature to this extent, before moving to query view based strategies.

Table 5.3: Average symmetric difference scores for top-10 results and conjunctive query processing (relative improvements with respect to the baseline algorithm are shown in the column Δ%; all improvements except (*)ed values are statistically significant)

%	TCP	DCP	aTCP	aDCP	TCP-QV	Δ%	DCP-QV	Δ%	aTCP-QV	Δ%	aDCP-QV	Δ%
10%	0.66	0.80	0.93	0.98	0.94	42%	0.98	23%	0.97	4%	0.98	0%*
20%	0.52	0.66	0.86	0.96	0.90	73%	0.95	44%	0.94	9%	0.96	0%*
30%	0.41	0.54	0.78	0.91	0.86	110%	0.92	70%	0.91	17%	0.93	2%
40%	0.32	0.43	0.70	0.85	0.84	163%	0.88	105%	0.87	24%	0.90	6%
50%	0.25	0.33	0.60	0.79	0.81	224%	0.84	155%	0.84	40%	0.86	9%
60%	0.19	0.25	0.52	0.71	0.79	316%	0.79	216%	0.79	52%	0.81	14%
70%	0.15	0.17	0.43	0.61	0.51	240%	0.58	241%	0.71	65%	0.73	20%

Our experiments reveal that for the conjunctive processing mode, TCP is the worst strategy. This is a rather expectable result as in an earlier study it is argued that for, say, two terms in a conjunctive query, TCP may have pruned a posting that is at the tail of one term's list and thus reduce the final rank of this posting which is at the top of the other term's list (see [55, Figure 1]). Furthermore, a TCP-like pruning strategy is also found less successful in ResIn framework [104]. This is attributed to the observation that the miss-queries are rather discriminative; i.e., return very few results. Recall that our test set also has similar properties to miss-queries, and the average result size is only 398. Indeed, we created another test set that includes the queries with the highest number of results in our collection and witnessed that TCP's performance can considerably improve. Nevertheless, in a typical setup with random queries, TCP is the worst performing algorithm for this case.

What is more surprising for conjunctive query processing case is the performance of the access-based strategies: aDCP and aTCP outperform TCP and DCP with a wide margin at all pruning levels. This is a new result that has not been reported before in the literature. We think that one reason of this great boost in performance may be the conjunctive processing of the training queries while computing the access counts. In the previous work, both training and testing have been conducted in disjunctive mode. We anticipate that the training in conjunctive mode more successfully distinguishes the documents that can also appear in the intersection of terms in other queries. Another remarkable issue is, our document-centric version of the access based strategy, aDCP, significantly

outperforms its term-centric adaptation. Indeed, aDCP achieves a similarity of 80% to the original results even when the index is halved, a striking success that has not been observed for any of the baseline algorithms even in the disjunctive case.

Turning our attention to the query view based strategies, we again report important improvements. This time, the worst performing strategies, TCP and DCP, have most benefited from the query views, even more than doubling or tripling their similarity scores at certain pruning levels. The gains on access-based strategies are less emphasized, though reaching to 40% and 9% at 50% pruning for aTCP-QV and aDCP-QV, respectively. Note that, aDCP reaches to very high similarity scores of 0.98 and 0.96 at 10% and 20% pruning levels, respectively; and these happen to be the only cases in Table 5.3 where the query view could not achieve any further improvements. For all other cases, query view based strategies again surpass their counterparts with a large margin, and reach to around 80% similarity level at a pruning level of 60%.

Effects of the training set size. For both query processing modes, we analyze how the performances of query view based strategies vary for training query sets with different characteristics. In Figure 5.2, we first consider the disjunctive case. In Figure 5.2(a) and (b), it is clearly seen that TCP-QV and DCP-QV improve proportionally to the training set size, respectively. Notably, even a training set of 10K queries improves performance in a statistically significant manner. As it can be anticipated from Table 5.1, the performance of 518K (1.8M) queries with top-10 results is slightly better than 10K (50K) queries with top-1000 results, respectively. For access-based strategies, to simplify the plots, we only provide sets with 1.8M queries with top-10 and 1000 results. For both cases (and other sets that are not shown here), the query view based strategies outperform their counterparts. Only for the smallest sets, namely 10K-top1000 and 50K-top1000, the improvements of aDCP-QV over aDCP are found not to be statistically significant.

In Figure 5.3 we demonstrate the behavior of the algorithms for the conjunctive processing mode. Again, TCP-QV and DCP-QV achieve higher scores with

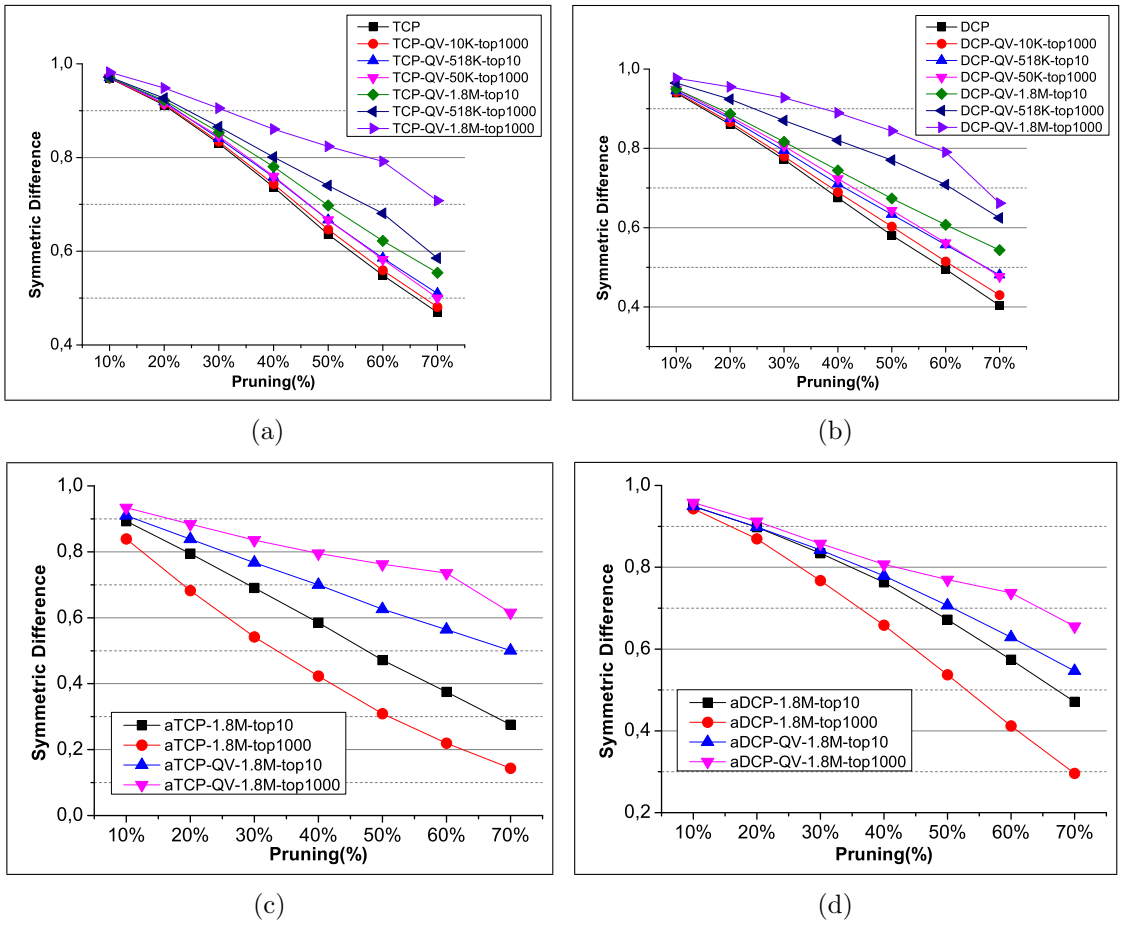


Figure 5.2: Effects of the training set size for disjunctive querying: (a) TCP vs. TCP-QV, (b) DCP vs. DCP-QV, (c) aTCP vs. aTCP-QV, and (d) aDCP vs. aDCP-QV.

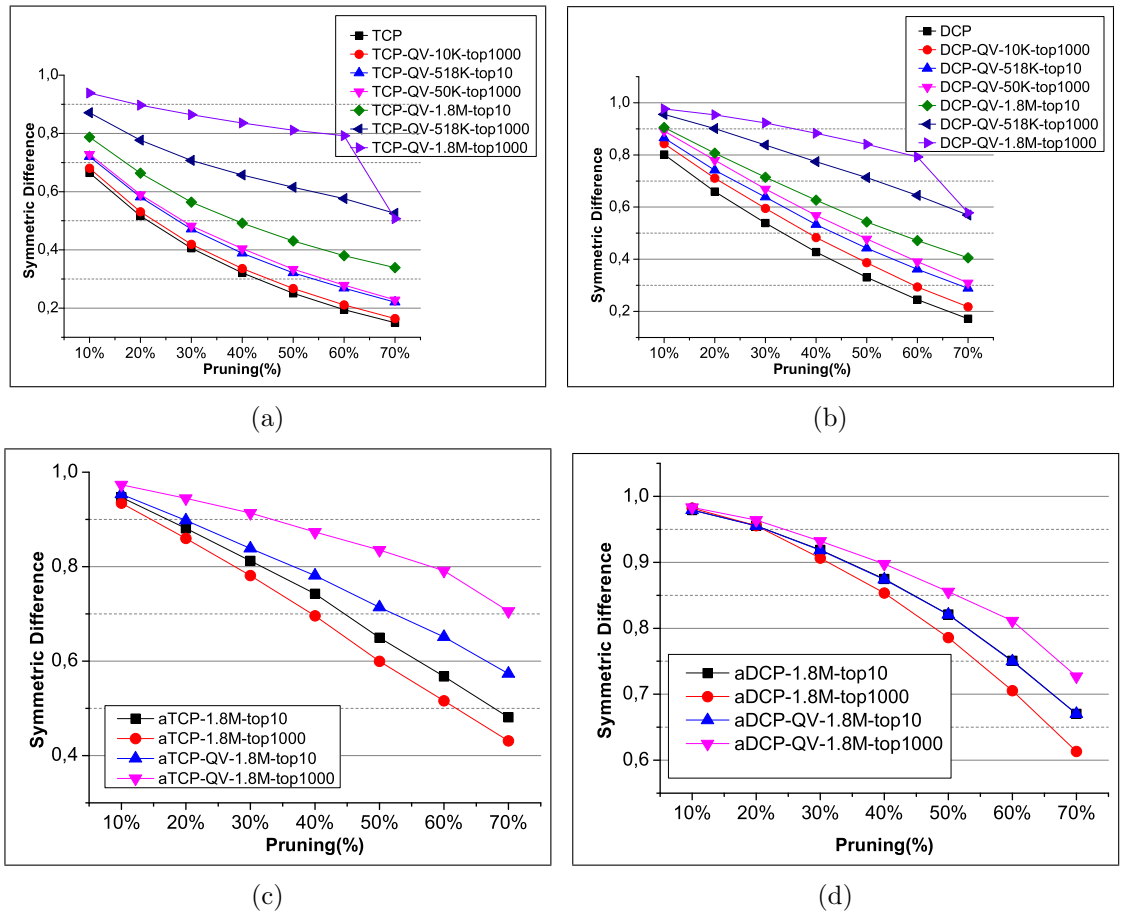


Figure 5.3: Effects of the training set size for conjunctive querying: (a) TCP vs. TCP-QV, (b) DCP vs. DCP-QV, (c) aTCP vs. aTCP-QV, and (d) aDCP vs. aDCP-QV.

the larger number of training queries. For aDCP-QV and aTCP-QV, trends are also similar, but for aDCP-QV the training set of 1.8M-top10 does not yield significantly different results from aDCP (as also seen from the overlapping lines in Figure 5.3(d). This implies that to further improve access-based strategies, training sets with larger number of queries or results should better be preferred for this query processing mode. We conclude that query view based strategies improve with larger train sets, but significant improvements are attainable by even using relatively smaller sets or larger sets with less number of results per query.

Effect of the test set size. We also conducted an experiment involving a set

of 100K random queries that is constructed as described in Section 5.5.1. To our knowledge, this is the largest set used for the evaluation of pruning performance. Due to time and resource limitations, this experiment is conducted for only 50% pruning level using 518K-top1000 training log in conjunctive processing mode. We compare our findings with those obtained for the same setup using test-1000 set. It turns our that (i) for each strategy, there is a slight increase in absolute symmetric difference scores but there is no statistically significant difference between their results on 1000 and 100K queries (except TCP-QV, of which absolute scores improve slightly more than the others on this experiment). The significance is computed using two-sample t-test (as sample sizes are different). Thus, the trends for each strategy can be concluded to be the same for both small and large test sets. (ii) According to one-way ANOVA (followed by Tukey test) and paired t-test analysis, query view based strategies again significantly outperform their baselines also for the 100K test set at 0.05 level. This means that trends and findings for the test-1000 set are also confirmed by the results obtained for the large test set.

Experiments for a ResIn-like framework. Up to here, we provide the effectiveness results assuming that the pruned index will replace the original index, say, at the back-end servers. As discussed before, an alternative use of a pruned index is locating it closer to the front-end, and directing only those queries that are not answered "correctly" (i.e., the same as the original index) to the backend server [104]. In this case, what is important is the number of queries which can be correctly answered by a pruned index. Conducting such an experiment would also make sense in our setup, since test-1000 set has similar characteristics to the miss-queries used in ResIn. As test-1000 queries do not appear in the training sets and appear only once in the log that is used to create test sets, they cannot be cached statically or dynamically, and would exactly constitute the miss-queries set for our setup. In Figure 5.4 , we show the number of correctly answered queries (i.e., for which, the symmetric difference score is 1[3]) for each case of Table 5.3 (i.e., for conjunctive mode). Clearly, for all cases, the query view based strategies considerably increase the number of queries with correct results. This

[3]Note that, we only consider whether top-10 results include the same documents for original and pruned cases, but disregard the order of documents.

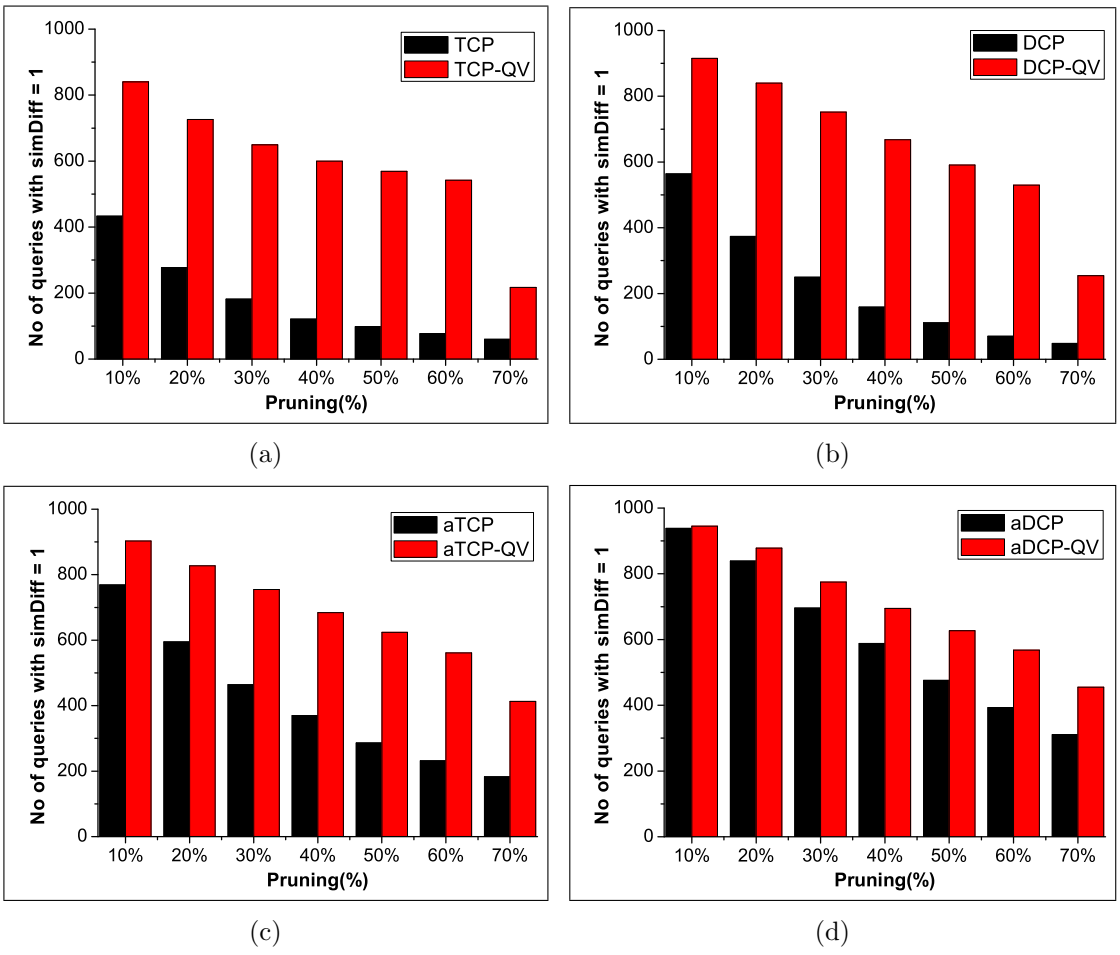


Figure 5.4: Number of queries with correct answers for pruning strategies and conjunctive mode: (a) TCP vs. TCP-QV, (b) DCP vs. DCP-QV, (c) aTCP vs. aTCP-QV, and (d) aDCP vs. aDCP-QV.

implies that, query view based strategies have a great potential to be employed in a ResIn-like framework to obtain higher performance. This is left as a future work.

5.5.3 Summary of the Findings

Our major results and contributions are summarized as follows:

- Using query views significantly improves all four pruning strategies for both disjunctive and conjunctive processing. The gains are proportional to the

number of queries and number of results retrieved per query in the training sets that are used to construct the query views. Though, it is still possible to obtain gains by using a smaller number of queries and larger number of results, or vice versa.

- Query view based strategies also increase the number of queries that are answered correctly by a pruned index. Thus, they allow a pruned index to be either used at the back-end providing higher effectiveness, or employed at the front end providing higher performance (i.e., as more queries will be satisfied at the front-end, less queries will be sent to the back-end).

- Access-based baseline strategies are inferior to TCP and DCP as shown before, but only for disjunctive querying. For the conjunctive case, which is the most crucial one for WSEs, we show that aTCP as proposed in [58] outperforms the methods that are not access-based. Furthermore, we present a new document-centric version of the algorithm, aDCP, which is superior to other three approaches; namely, TCP, DCP and aTCP.

- We describe a carefully tailored experimental framework that is reliable for extensive testing. Our query set has realistic characteristics that can be observed in a WSE setup [104]. Our gains are obtained for such a set and verified using the statistical tests. Furthermore, a large set of 100K queries is used for a subset of the experiments and also found to exhibit exactly the same trends. To our knowledge, this is the largest set used in an index pruning experiment (i.e., an order of magnitude larger than the set employed in [104]).

- At last, we compare and evaluate static pruning algorithms in a unified framework for both modes of querying. This has not been done for all of these approaches before (except [9], which compares TCP and DCP only for disjunctive mode).

5.6 Conclusions and Future Work

In this chapter, we propose query view based strategies for static pruning to improve the top-ranked result quality. We incorporate query views into a number of strategies that exist in the literature, and show that the new strategies considerably outperform their counterparts especially for the higher levels of pruning.

As a future work, we first aim to use the frequencies of terms in the query views to further improve our strategies. Another promising direction is using the query view heuristic in dynamic pruning, in a similar manner to [59, 60].

Chapter 6

Conclusions and Future Work

Devising efficient methods for each fundamental component, namely; crawler, indexer and query processor, in a Web search engine is an important research topic. In this thesis, for each one of these components, we proposed some efficient strategies that may be applicable especially when a grouping of documents in its broadest sense (i.e., in terms of automatically obtained classes/clusters, or manually edited categories) is available. We also exploited query views that are based on the search engine query logs to tailor more effective static pruning techniques.

More specifically, for the purposes of focused crawling —a paradigm that is essentially employed in vertical search engines, we proposed a rule-based strategy. In this case, the rules represented the linkage relationships among the document classes in a taxonomy. This approach remedied an important weakness of a pioneering focused crawling strategy in the literature; i.e., by combining rules, the rule-based strategy becomes capable of reaching relevant pages through a path of irrelevant pages (an effect known as tunneling). In the experiments, our crawler was observed to be more successful in finding relevant (on-topic) pages in comparison to a baseline focused crawler; a result justifying our intuition.

In this thesis, document clusters and categories are also employed for improving search performance. In particular, we discussed possible query processing methods for typical cluster-based retrieval (CBR). We introduced a new index organization, so-called cluster-skipping inverted index structure (CS-IIS) which

blends cluster and document information and allows faster query processing. We evaluated our approach in an extensive experimental setup involving automatically clustered and manually categorized datasets and with several parameters. We showed that typical-CBR with CS-IIS outperforms other query evaluation strategies using ordinary index files.

We further enhanced CS-IIS so that all information to compute query-cluster similarities during query evaluation is stored in a single index file. We introduced an incremental-CBR strategy that operates on top of this new index structure, and demonstrated its search efficiency in an environment where all index files are compressed, a typical situation for real life search engines.

Our results for searching document groups are remarkable in the following sense. We show that search using document clusters or categories can provide significant efficiency improvements (especially in terms of in-memory execution time) while yielding query result with a quality as good as that obtained over the entire collection; i.e., without any sort of grouping. Our approaches are applicable in the scenarios where the data is inherently categorized (such as a Web directory) or an automatic clustering of collection is possible by some means. In the latter case, the clustering structure can be created with respect to actual document contents, which is more feasible with medium-scale collections; or some other basic feature, such as the website of a document, as we exemplified in this thesis.

Finally, we made use of search engine logs to develop better strategies for static index pruning. In particular, query view approach was incorporated into a set of existing pruning strategies, as well as some new variants proposed by us. Query view based strategies significantly outperformed the baseline approaches from literature in terms of the query output quality, for both disjunctive and conjunctive evaluation of queries. This is an important result, as the latter two types of queries constitute the majority of queries submitted to search engines. Our results also implied that the index files pruned by query view based strategies can either replace the original index at the back-end, or serve as a list cache at the front-end of a search engine.

There are many future work directions regarding the contributions of this

thesis. The rule-based focused crawler can be further enhanced using rules that are obtained by more-sophisticated techniques from machine learning and data mining literature. We believe that search using document clusters/categories is an issue that deserves more attention. The performance of CS-IIS can be investigated in an environment with list caching for more realistic applications. In this sense, an interesting direction is considering the performance of CS-IIS based retrieval approaches when only certain blocks of the posting lists (corresponding to, say, most popular clusters) are fetched from the disk and cached. It is also possible to apply proposed CBR strategies in a framework of patent retrieval. Our research for the latter topic is already underway. Finally, query views can be used for index pruning in more sophisticated ways (e.g., by considering the access frequencies in query views) and for dynamic pruning purposes. These latter issues are also included in our current research agenda.

Bibliography

[1] C. C. Aggarwal, F. Al-Garawi, and P. S. Yu. Intelligent crawling on the world wide web with arbitrary predicates. In *Proceedings of the 10th International Conference on World Wide Web*, pages 96–105, 2001.

[2] I. S. Altingovde, F. Can, and Ö. Ulusoy. Algorithms for within-cluster searches using inverted files. In A. Levi, E. Savas, H. Yenigün, S. Balcisoy, and Y. Saygin, editors, *ISCIS*, volume 4263 of *Lecture Notes in Computer Science*, pages 707–716. Springer, 2006.

[3] I. S. Altingovde, F. Can, and Ö. Ulusoy. Efficient processing of category-restricted queries for Web directories. In C. Macdonald, I. Ounis, V. Plachouras, I. Ruthven, and R. W. White, editors, *ECIR*, volume 4956 of *Lecture Notes in Computer Science*, pages 695–699. Springer, 2008.

[4] I. S. Altingovde, E. Demir, F. Can, and Ö. Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Transactions on Information Systems*, 26(3):1–36, 2008.

[5] I. S. Altingovde, E. Demir, F. Can, and Ö. Ulusoy. Site-based dynamic pruning for query processing in search engines. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 861–862, 2008.

[6] I. S. Altingovde, R. Ozcan, S. Cetintas, H. Yilmaz, and Ö. Ulusoy. An automatic approach to construct domain-specific Web portals. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management*, pages 849–852, 2007.

[7] I. S. Altingovde, R. Ozcan, H. C. Ocalan, F. Can, and Ö. Ulusoy. Large-scale cluster-based retrieval experiments on Turkish texts. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 891–892, 2007.

[8] I. S. Altingovde, R. Ozcan, and Ö. Ulusoy. Exploiting query views for static index pruning in Web search engines. In *Proceedings of the Eighteenth ACM Conference on Information and Knowledge Management*, 2009. To appear.

[9] I. S. Altingovde, R. Ozcan, and Ö. Ulusoy. A practitioner's guide for static index pruning. In M. Boughanem, C. Berrut, J. Mothe, and C. Soulé-Dupuy, editors, *ECIR*, volume 5478 of *Lecture Notes in Computer Science*, pages 675–679. Springer, 2009.

[10] I. S. Altingovde and Ö. Ulusoy. Exploiting interclass rules for focused crawling. *IEEE Intelligent Systems*, 19(6):66–73, 2004.

[11] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, New York, USA, 1973.

[12] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42, 2001.

[13] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[14] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 226–233, 2005.

[15] 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, 2006.

[16] I. Antonellis, H. Garcia-Molina, and J. Karim. Tagging with queries: how and why? In R. A. Baeza-Yates, P. Boldi, B. A. Ribeiro-Neto, and B. B. Cambazoglu, editors, *WSDM (Late Breaking-Results)*, 2009.

[17] R. Baeza-yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed Web retrieval. In *Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[18] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 183–190, 2007.

[19] A. A. Barfourosh, M. L. Anderson, D. Perlis, and H. M. Nezhad. Information retrieval on the World Wide Web and active logic: A survey and problem definition. Technical Report CS-TR-4291, Computer Science Dept., University of Maryland, 2002.

[20] R. Blanco. *Index Compression for Information Retrieval Systems.* PhD thesis, University of A Coruna, Spain, 2008.

[21] R. Blanco and A. Barreiro. Boosting static pruning of inverted files. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 777–778, 2007.

[22] R. Blanco and A. Barreiro. Static pruning of terms in inverted files. In *Proceedings of the 29th European Conference on IR Research*, pages 64–75, 2007.

[23] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proceedings of the Data Compression Conference*, pages 342–351, Washington, DC, 2002.

[24] P. D. Bra, G.-J. Houben, Y. Kornatzky, and R. Post. Information retrieval in distributed hypertexts. In *Proceedings of the 4th RIAO Conference*, pages 481–493, 1994.

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

[26] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 30–38, 1995.

[27] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97–110, 1985.

[28] C. Buckley and E. M. Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 33–40, 2000.

[29] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 182–189, 2006.

[30] F. Cacheda and R. A. Baeza-Yates. An optimistic model for searching Web directories. In S. McDonald and J. Tait, editors, *ECIR*, volume 2997 of *Lecture Notes in Computer Science*, pages 364–377, 2004.

[31] F. Cacheda, V. Carneiro, C. Guerrero, and Á. Viña. Optimization of restricted searches in Web directories using hybrid data structures. In F. Sebastiani, editor, *ECIR*, volume 2633 of *Lecture Notes in Computer Science*, pages 436–451, 2003.

[32] B. B. Cambazoglu and C. Aykanat. Performance of query processing implementations in ranking-based text retrieval systems using inverted indices. *Information Processing and Management*, 42(4):875–898, 2006.

[33] B. B. Cambazoglu, E. Karaca, T. Kucukyilmaz, A. Turk, and C. Aykanat. Architecture of a grid-enabled Web search engine. *Information Processing and Management*, 43(3):609–623, 2007.

[34] B. B. Cambazoğlu. *Models and Algorithms for Parallel Text Retrieval*. PhD thesis, Bilkent University, Ankara, Turkey, 2006.

[35] F. Can. Incremental clustering for dynamic information processing. *ACM Transactions on Information Systems*, 11(2):143–164, 1993.

[36] F. Can. On the efficiency of best-match cluster searches. *Information Processing and Management*, 30(3):343–362, 1994.

[37] F. Can, I. S. Altingovde, and E. Demir. Efficiency and effectiveness of query processing in cluster-based retrieval. *Information Systems*, 29(8):697–717, 2004.

[38] F. Can, E. A. Fox, C. D. Snavely, and R. K. France. Incremental clustering for very large document databases: initial MARIAN experience. *Information Sciences*, 84(1&2):101–114, 1995.

[39] F. Can, S. Kocberber, E. Balcik, C. Kaynak, H. C. Ocalan, and O. M. Vursavas. First large-scale information retrieval experiments on Turkish texts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 627–628, 2006.

[40] F. Can, S. Kocberber, E. Balcik, C. Kaynak, H. C. Ocalan, and O. M. Vursavas. Information retrieval on Turkish texts. *Journal of the American Society for Information Science*, 59(3):407–421, 2008.

[41] F. Can and E. A. Ozkarahan. Similarity and stability analysis of the two partitioning type clustering algorithms. *Journal of the American Society for Information Science*, 36(1):3–14, 1985.

[42] F. Can and E. A. Ozkarahan. Concepts and effectiveness of the cover-coefficient based clustering methodology for text databases. *ACM Transactions on Database Systems*, 15(4):483–517, 1990.

[43] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–50, 2001.

[44] M. Castellanos. Hotminer: Discovering hot topics from dirty text. In M. W. Berry, editor, *Survey of Text Mining*, pages 223–233. Springer-Verlag, 2003.

[45] S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, San Francisco, CA, USA, 2003.

[46] S. Chakrabarti, M. Joshi, K. Punera, and D. M. Pennock. The structure of broad topics on the Web. In *Proceedings of the 11th International Conference on World Wide Web*, pages 251–262, 2002.

[47] S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proceedings of the 11th International Conference on World Wide Web*, pages 148–159, 2002.

[48] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[49] M. Chau and H. Chen. Personalized and focused Web spiders. In N. Zhong, J. Liu, and Y. Yao, editors, *Web Intelligence*, pages 197–217. Springer-Verlag, 2003.

[50] F. Chierichetti, A. Panconesi, P. Raghavan, M. Sozio, A. Tiberi, and E. Upfal. Finding near neighbors through cluster pruning. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 103–112, 2007.

[51] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1-7):161–172, 1998.

[52] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[53] W. Croft. A model of cluster searching based on classification. *Information Systems*, 5(3):189–195, 1980.

[54] E. S. de Moura, C. F. dos Santos, B. Araujo, A. S. Silva, P. Calado, and M. A. Nascimento. Locality-based pruning methods for Web search. *ACM Transactions on Information Systems*, 26(2):1–28, 2008.

[55] E. S. de Moura, C. F. dos Santos, D. R. Fernandes, A. S. Silva, P. Calado, and M. A. Nascimento. Improving Web search efficiency via a locality based static pruning method. In *Proceedings of the 14th International Conference on World Wide Web*, pages 235–244, 2005.

[56] M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 527–534, 2000.

[57] A. El-Hamdouchi and P. Willett. Techniques for the measurement of clustering tendency in document retrieval systems. *Journal of Information Science*, 13(6):361–365, 1987.

[58] S. Garcia. *Search Engine Optimization Using Past Queries*. PhD thesis, RMIT University, Melbourne, Australia, 2007.

[59] S. Garcia and A. Turpin. Efficient query evaluation through access-reordering. In H. T. Ng, M.-K. Leong, M.-Y. Kan, and D. Ji, editors, *AIRS*, volume 4182 of *Lecture Notes in Computer Science*, pages 106–118. Springer, 2006.

[60] S. Garcia, H. E. Williams, and A. Cannane. Access-ordered indexes. In *Proceedings of the 27th Australasian Conference on Computer Science (ACSC)*, pages 7–14, 2004.

[61] D. Harman. Ranking algorithms. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 363–392. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[62] M. A. Hearst and J. O. Pedersen. Reexamining the cluster hypothesis: scatter/gather on retrieval results. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 76–84, 1996.

[63] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 3rd edition. Morgan Kaufmann Publishers, San Mateo, CA, 2002.

[64] M. Hersovici, M. Jacovi, Y. S. Maarek, D. Pelleg, M. Shtalhaim, and S. Ur. The shark-search algorithm —an application: tailored Web site mapping. *Computer Networks and ISDN Systems*, 30(1-7):317–326, 1998.

[65] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[66] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[67] N. Jardine and C. J. van Rijsbergen. The use of hierarchic clustering in information retrieval. *Information Storage and Retrieval*, 7(5):217–240, 1971.

[68] R. B. Kellogg and M. Subhas. Text to hypertext: can clustering solve the problem in digital libraries? In *Proceedings of the First ACM International Conference on Digital Libraries*, pages 144–150, 1996.

[69] S. Kocberber, F. Can, and J. M. Patton. Optimization of signature file parameters for database with varying record lengths. *The Computer Journal*, 42(1):11–23, 1999.

[70] Y. Kural, S. Robertson, and S. Jones. Deciphering cluster representations. *Information Processing and Management*, 37(4):593–601, 2001.

[71] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In A. H. H. Ngu, M. Kitsuregawa, E. J. Neuhold, J.-Y. Chung, and Q. Z. Sheng, editors, *WISE*, volume 3806 of *Lecture Notes in Computer Science*, pages 470–477, 2005.

[72] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing Web pages by "information unit". In *Proceedings of the 10th International Conference on World Wide Web*, pages 230–244, 2001.

[73] X. Liu and W. B. Croft. Cluster-based retrieval using language models. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 186–193, 2004.

[74] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 129–140, 2003.

[75] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[76] A. Mccallum, K. Nigam, J. Rennie, and K. Seymore. Building domain-specific search engines with machine learning techniques. In *Proceedings of AAAI Spring Symp. Intelligent Agents in Cyberspace*, pages 28–39, 1999.

[77] A. K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. `http://www.cs.cmu.edu/~mccallum/bow`, 1996.

[78] F. Menczer, G. Pant, and P. Srinivasan. Topical Web crawlers: evaluating adaptive algorithms. *ACM Transactions on Internet Technology*, 4(4):378–419, 2004.

[79] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[80] S. Mukherjea. WTMS: a system for collecting for collecting and analyzing topic-specific Web information. *Computer Networks*, 33(1-6):457–471, 2000.

[81] G. Muresan and D. J. Harper. Topic modeling for mediated access to very large document collections. *Journal of the American Society for Information Science*, 55(10):892–910, 2004.

[82] D. M. Murray. *Document Retrieval Based on Clustered Files*. PhD thesis, Cornell University, Ithaca, NY, USA, 1972.

[83] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 191–198, 2007.

[84] R. Nuray and F. Can. Automatic ranking of information retrieval systems using data fusion. *Information Processing and Management*, 42(3):595–614, 2006.

[85] ODP. Open directory project. `http://www.dmoz.org`, 2009.

[86] G. Pant and P. Srinivasan. Learning to crawl: comparing classification schemes. *ACM Transactions on Information Systems*, 23(4):430–462, 2005.

[87] G. Pant and P. Srinivasan. Link contexts in classifier-guided topical crawlers. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):107–122, 2006.

[88] M. T. Pazienza, S. A., and M. Vindigni. Purchasing the Web: an agent based e-retail system with multilingual knowledge. In *Proceedings of WI2003 Workshop on Applications, Products and Services of Web-based Support Systems*, 2003.

[89] M. Persin. Document filtering for fast ranking. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 339–348, 1994.

[90] 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.

[91] B. Poblete and R. Baeza-Yates. Query-sets: using implicit feedback and query patterns to organize Web documents. In *Proceeding of the 17th International Conference on World Wide Web*, pages 41–50, 2008.

[92] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st International Conference on Scalable Information Systems*, page 34, 2006.

[93] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. 3rd edition. McGraw-Hill Science/Engineering/Math, 2002.

[94] G. Salton. Cluster search strategies and the optimization of retrieval effectiveness. In G. Salton, editor, *The SMART Retrieval System – Experiments in Automatic Document Processing*, pages 223–242. Prentice Hall, Englewood Cliffs, NJ, 1971.

[95] G. Salton. *Dynamic Information and Library Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1975.

[96] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, 1989.

[97] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[98] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[99] J. Shakes, M. Langheinrich, and O. Etzioni. Dynamic reference sifting: a case study in the homepage domain. In *Proceedings of the Sixth International Conference on World Wide Web*, pages 1193–1204, 1997.

[100] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large Web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[101] F. Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on IR Research*, pages 101–112, 2007.

[102] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 305–312, 2004.

[103] S. Sizov, J. Graupmann, and M. Theobald. From focused crawling to expert information: an application framework for Web exploration and portal generation. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1105–1108, 2003.

[104] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: a combination of results caching and index pruning for high-performance Web search engines. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 131–138, 2008.

[105] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 175–182, 2007.

[106] T. T. Tang, D. Hawking, N. Craswell, and K. Griffiths. Focused crawling for both topical relevance and quality of medical information. In *Proceedings of the 14th ACM International Conference on Information and Knowledge management*, pages 147–154, 2005.

[107] A. Tombros. *The Effectiveness of Query-Based Hierarchic Clustering of Documents for Information Retrieval*. PhD thesis, University of Glasgow, Glasgow, UK, 2002.

[108] TREC. Text retrieval conference homepage. `http://trec.nist.gov`, 2009.

[109] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management*, pages 987–990, 2007.

[110] C. J. van Rijsbergen. *Information Retrieval*. 2nd edition. Butterworth, 1979.

[111] E. M. Voorhees. The cluster hypothesis revisited. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 188–196, 1985.

[112] E. M. Voorhees. *The Effectiveness and Efficiency of Agglomerative Hierarchic Clustering in Document Retrieval*. PhD thesis, Cornell University, Ithaca, NY, USA, 1986.

[113] E. M. Voorhees. The efficiency of inverted index and cluster searches. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 164–174, 1986.

[114] WEBBASE. Stanford University WebBase project website. `http://diglib.stanford.edu:8091/~testbed/doc2/WebBase/`, 2007.

[115] W. Webber and A. Moffat. In search of reliable retrieval experiments. In *Proceedings of the 10th Australasian Document Computing Symposium (ADCS)*, pages 26–33, 2005.

[116] P. Willett. Recent trends in hierarchic document clustering: a critical review. *Information Processing and Management*, 24(5):577–597, 1988.

[117] P. Willett, V. Winterman, and D. Bawden. Implementation of nonhierarchic cluster analysis methods in chemical information structure search. *Journal of Chemical Information and Computer Sciences*, 26(3):109–118, 1986.

[118] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. 2nd edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[119] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.

[120] C. T. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[121] Zettair. Zettair open-source search engine, designed and written by the search engine group at rmit university. `http://www.seg.rmit.edu.au/zettair/`, 2006.

[122] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.