Collection Selection for Distributed Web Search

Using Highly Discriminative Keys, Query-driven Indexing and ColRank

By Sander Bockting

Master's Thesis
University of Twente
February 2009

Graduation committee

Chairman:	Dr.ir. Djoerd Hiemstra
1st coordinator:	Pavel V. Serdyukov, MSc.
2nd coordinator:	Almer S. Tigelaar, MSc.

To my parents, who have always supported me.

Summary

Current popular web search engines, such as Google, Live Search and Yahoo!, rely on crawling to build an index of the World Wide Web. Crawling is a continuous process to keep the index fresh and generates an enormous amount of data traffic. By far the largest part of the web remains unindexed, because crawlers are unaware of the existence of web pages and they have difficulties crawling dynamically generated content.

These problems were the main motivation to research distributed web search. We assume that web sites, or peers, can index a collection consisting of local content, but possibly also content from other web sites. Peers cooperate with a broker by sending a part of their index. Receiving indices from many peers, the broker gains a global overview of the peers' content. When a user poses a query to a broker, the broker selects a few peers to which it forwards the query. Selected peers should be promising to create a good result set with many relevant documents. The result sets are merged at the broker and sent to the user.

This research focuses on *collection selection*, which corresponds to the selection of the most promising peers. The use of *highly discriminative keys* is employed as a strategy to select those peers. A highly discriminative key is a term set which is an index entry at the broker. The key is highly discriminative with respect to the collections because the posting lists pointing to the collections are relatively small. *Query-driven indexing* is applied to reduce the index size by only storing index entries that are part of popular queries. A PageRank-like algorithm is also tested to assign scores to collections that can be used for ranking.

The *Sophos* prototype was developed to test these methods. Sophos was evaluated on different aspects, such as collection selection performance and index sizes. The performance of the methods is compared to a baseline that applied language modeling onto merged documents in collections. The results show that Sophos can outperform the baseline with ad-hoc queries on a web based test set. Query-driven indexing is able to substantially reduce index sizes against a small loss in collection selection performance. We also found large differences in the level of difficulty to answer queries on various corpus splits.

Preface

My first steps in the field of Information Retrieval were with a course of the same name. During that course, Matthijs Ooms and I developed an image retrieval system for young children and wrote a paper about its evaluation. Djoerd encouraged me to improve the paper to get it accepted for a conference, resulting in my first publication and my first visit to a conference.

During this conference, Djoerd and I were walking through the streets of Maastricht when heavy rainfall set in. We took shelter and started talking about my career plans. That conversation changed my view on academic research, making it a serious option as a next career step. Being at that conference also had a strong motivational impact in completing this research. I would like to express my gratitude to Djoerd for his big and positive role in the last part of my study.

I would also like to thank my other two supervisors for their added value. I have got to know Pavel as a knowledgeable person with great insight in Information Retrieval. And, although I know Almer for a relatively short but enjoyable period of time, his quick and direct feedback really improved this thesis.

This research benefited from discussions with Harold van Heerde, Matthijs Ooms, Robin Aly and Viet Yen Nguyen, but most notably Simon de Hartog. Simon also taught me some lessons in efficient coding; a coding match for a small tool was miserably lost by me (C#) in favor of Simon (C++). Special thanks go out to Ander de Keijzer, who put his machine at my disposal so that I could run my experiments.

I would also like to thank my fellow lab rats: Bertold, Erwin, Kien, Matthijs (Abma and Ooms), Maurits, Michiel, Peter, Remko and Sven. They were always in for serious helpful talk or some relaxing fun chat. I will miss those conversations, but most of all, I will miss our pleasant “4-uur Cup-a-Choco”.

This thesis would never have existed without the unconditional support of my parents Jos and Yvonne: mom, dad, thank you from the bottom of my heart.

Enschede, February 2009 *Sander Bockting*

Contents

1

Introduction

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it.

– Dr. Samuel Johnson (1709–1784)

1.1 Information retrieval

1.1.1 General introduction

Nowadays, almost everybody has used Internet search engines. Although the goals may vary widely – obtaining stock information, writing a paper or just getting some recipes for dinner – all users have something in common: they start with an *information need*. A user can describe the information need by formulating a *query*. A query is a set of terms, or keywords, which enables a search engine to find *relevant* documents that help fulfilling the users' information need. Generally, it does so by matching the query terms to the words contained in the documents. Instead of comparing queries with all documents, an *index* is distilled from all documents in the corpus and this is used for the matching process. This index is comparable to the index in the back of a book. Models and techniques have been developed to optimize the retrieval of those documents in the corpus that best answer the queries. Using those models, the result set of retrieved documents is *ranked* such that the most relevant documents are put on top of the result list.

A popular approach for ranking is looking at the frequency of term occurrences in documents of a collection. If a query term occurs frequently in a document, that document can be a relevant document. However, if that query term occurs in many documents of the collection, the specific documents may become less relevant. The latter concept is often referred to as *inverse document frequency* and is calculated by dividing the total number of documents in the collection with the number of documents in which the query term occurs. By taking the logarithm of this fraction, and multiplying this with a (normalized) *term frequency*, a reasonable ranking mechanism has been created. The combination of term frequency and inverse document frequency is often referred to as TF.IDF [74]. It ranks documents with high term frequencies higher when no other documents have high frequencies of those terms. More sophisticated approaches are also used, such as counting the number of links referring to web

pages or language models that calculate the probability that a document could have generated the query terms. Hiemstra gives a more elaborate description of Information Retrieval and frequently used models for information retrieval [33].

1.1.2 Centralized search

Search engines like Google, Yahoo! and Live Search are examples of centralized search engines. Their approach amounts to crawling, indexing and searching. Crawling is a systematic approach of visiting and copying web pages from the Internet. All downloaded pages are stored in centralized data centers. There, the downloaded pages are indexed and stored on index servers. When the user searches by posing a query, the query is sent to the index server. There, a lookup is performed in the index on the query terms. The result of this lookup is sent to the document servers that actually retrieve the documents. This is done to generate snippets that shortly describe the results. These snippets are sent to the user as the query result.

This approach of search is called centralized, because all activities (crawling, indexing and searching) are controlled and performed at locations where all systems are clustered, and the number of locations is relatively limited. Just to give an impression: according to estimates, Google uses 450,000 servers racked up in thousands of clusters in dozens of data centers around the world [15]. These servers are comparable to the average consumer PC.[1]

1.1.3 Problems with centralized search

The amount of data on the web is estimated to grow exponentially [78]. This growth poses problems for traditional, centralized web search engines. The continuously changing and growing data requires frequent visits by crawlers, just to keep the index fresh. This crawling should be done often, but generates a huge amount of traffic, making it impossible to do frequent crawls. With an estimated four weeks update interval, updates are performed relatively slow [60, 81].

Besides index freshness problems, it is simply impossible to index everything. In 1999, the estimated indexable web size was 800 million pages [41]. The indexable web, also known as the visible web, consists of web pages that are accessible by search engines. In 2002, this estimate increased to 8.9 billion pages [45]. In January 2005, the visible web was estimated to be 11.5 billion pages [30]. These are large numbers, but they are tiny when compared to the estimated deep web size: 4,900 billion pages [45]. The deep web contains items such as databases and dynamic web sites. In November 2004, Google indexed 8 billion pages[2] which is a tiny number, compared to the total estimated web size.

[1] More technical information on Google is available at http://www.google.nl/intl/en/corporate/tech.html.

[2] Claimed on http://www.google.com/intl/en/corporate/history.html as of April 2008.

Centralized search engines also introduce social problems. They have a monopoly in search, almost entirely controlling web search and largely influencing what can be found on the Internet. As centralized web search engines require large server farms, it is unlikely that new competitors will enter the search market, unless a different approach will produce promising results.

1.1.4 Distributed information retrieval

To search in a decentralized fashion, the centralized control will have to be removed and different actions have to be performed in a distributed fashion. Callan [12] identified a set of problems that need to be addressed to enable decentralization of search. The identified problems came to be known as *distributed information retrieval* and comprise the following problems:

Resource description: Instead of downloading the collections of documents (websites) to a centralized location, documents are left at their place. To be able to search those documents, a description of each of the document collections will have to be made.

Resource selection: When a user has formulated a query, the document collections will have to be found that are *most promising* for answering that query. The resource descriptions for each collection can be used for that purpose. In this research, a resource is defined as a collection of documents; resource selection and collection selection are interchangeable concepts.

Result merging: Once the most promising collections have been identified, the query is issued on those collections to find the best matching documents from those collections. Each of the collections produces a result list, which will have to be merged and ranked.

Distributed information retrieval solves the main problems of centralized search by removing the need to crawl the entire document collection. Each document collection can be indexed frequently – possibly whenever something changes – and it can be indexed better. For example, it could be possible to index relational databases as local indexers can have direct access to this type of information. It seems to be a promising approach, under the reservations that a scalable solution can be built that yields good retrieval performance.

The next section describes a scenario of a distributed information retrieval system. This scenario will act as a basis for this thesis in evaluating the retrieval performance of such a system.

1.2 Scenario

Figure 1.1 shows three types of entities: peers, brokers and clients. Every peer runs a web server with a search engine. The search engine indexes the local website(s), but is not limited to those as it may also index other websites. There can be many millions of peers. When a user has an information need, he can

pose a query at the client. The client sends the query to the broker. In response to that, the broker tries to identify the peers that seem to be the most promising in answering that query. This will be a small amount of peers, e.g. five to ten peers. The query is routed to those peers and the results are returned to the broker. The broker merges the results and sends it to the client. There is a limited amount of brokers, e.g. one per world continent. The brokers can operate independently from each other.

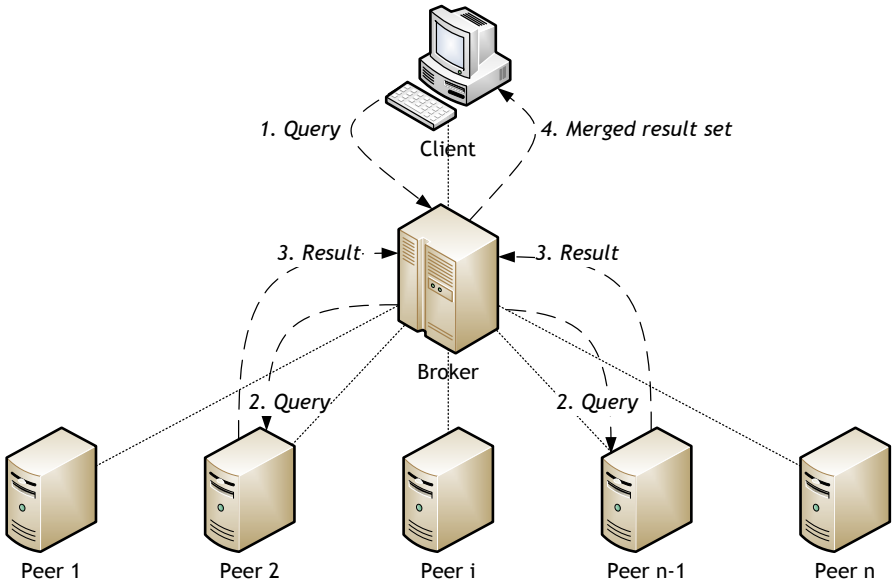

Fig. 1.1. Peers are accessible via a broker to answer queries from clients

Cooperation between peers and brokers is required to enable brokers to identify most promising peers. Therefore, every peer sends a small part of its index to the broker. This part of the index cannot be too small, as this renders it impossible to make good judgments about the ability of a peer to satisfactory answer queries. However, it also cannot be too large, as this imposes scalability issues in terms of data traffic when scaling the sketched scenario to the entire web. A balance between the two extremes is required. The next section describes a technique to balance the amount of index data sent to the broker.

1.3 Short introduction to highly discriminative keys

A book index can be created by writing down all interesting words, together with the page numbers where every word occurs. When this is done for every word, it is possible to retrieve every word occurrence in the book. When looking for

multiple terms, every term can be looked up and the intersection of the page number sets gives the pages where all terms occur on one page.

Instead of indexing pages of a book, a similar approach can be used for collections in a corpus, resulting in an index for the corpus. An index that is created in this way is known as a *standard inverted index*. There are a number of disadvantages to this approach:

- There may be terms that are rarely queried for. These terms are probably infrequent, so the list of occurrences of those terms is not so large for every term. However, the sum of all those terms may add up noticeable in the index.
- Multi-term queries will always require an intersection with single term occurrences. This is usually slower than doing this with an index that also contains multi-term entries. It also requires storage of all term occurrences for the intersection to work.

Another negative consequence is that the index will become large, which is undesirable when it should be sent to a broker.

It is possible to reduce the index size by only storing entries that are useful. *Highly discriminative keys* (HDKs) are keys that may consist of one or more terms and are chosen in such a way that they only select a limited number of documents, i.e. they are discriminative for collections in the corpus [44]. For example, 'red' and 'car' are both less discriminative than 'red car'. Only storing the locations that contain 'red car' will result in a smaller index than storing the locations that contain 'red' or 'car'. The amount of discrimination for a key (the number of stored locations) acts as a way to regulate the index size. When a key selects too many collections, it is discarded as a highly discriminative key at the broker, as the global index would otherwise become too large. This is comparable to TF.IDF working at peer level.

Highly discriminative keys will be explained in depth in the next chapter and are a pivotal concept in this research.

1.4 Research scope

This research will focus on resource description and resource selection using HDKs. Indices will be built to estimate which collections will be most promising for answering queries. This research will look at the quality of those estimates. It will not investigate actually sending the query to those collections to get a ranked list of results, i.e. result merging, as described in section 1.1.4. Neither will the focus of this research be on the efficiency of selecting the collection. Although efficiency is important as well – an unscalable solution has only limited applicability, namely for small collections – this research will first look at whether the techniques are worth optimizing at all.

1.5 Research questions

Section 1.1.3 described several problems with centralized search. An efficient implementation of the scenario from Section 1.2 may solve those problems. HDKs may provide the required efficiency. By removing the need for crawling, building compact indices at each peer and sending those indices to one broker, distributed information retrieval may become a promising alternative to centralized search. This research will investigate the feasibility of such an approach by looking at the quality of collection selection. This will be done by combining HDKs with several popular information retrieval techniques.

One of the techniques is query-driven indexing, which prunes the index by removing all index entries that are not queried for. These entries are identified by analysis of a search engine query log. This helps in reducing the index size and communication overhead, hopefully without affecting retrieval performance.

Another technique is a distributed variant of Google PageRank [10]. PageRank assigns importance judgments to web pages by counting the number of hyperlinks referring to that page. More hyperlinks referring to a page implies higher importance, which can be used to rank the results. A comparable algorithm can be created to assign importance judgments to collections, instead of pages. Such an algorithm will be tested in this research.

In earlier research, HDKs pointed to a number of documents in one or more collections [44, 51]. In the scenario from Section 1.2, storing the pointers to the actual documents is not required, as peers fulfill the role of finding the actual documents. It suffices to store the number of key occurrences at a peer. We propose the novel technique of applying HDKs for collection selection; every HDK that is stored at the broker contains a list of peers with the number of key occurrences at those peers. This results in the following research question:

> *In which ways can web search be improved with collection selection from distributed information retrieval using a combination of highly discriminative keys and 1) query-driven indexing, 2) distributed PageRank?*

An obvious performance indicator for improvement is looking at the quality of the selected collections. A large set of documents, a *corpus*, can be split in several collections. A *collection* is a subset of the documents in the corpus. The way the corpus is split in collections affects the distribution of relevant documents and on the ability to select the collections with the most relevant documents. This leads to the following question:

> *Research question 1: What is the effect of splitting the document corpus?*

Five scenarios of document corpus splits will be analyzed: per peer (by IP address), clusters of peers, overlapping collections (adding duplicates), and two splits with either a large or small amount of randomly created collections.

Given a query, collection selection is all about ranking the collections in such a way that the collection with the largest amount of relevant documents is ranked

on top. This is important, because the user query is routed to a small amount of peers to get an answer. This results in the following question:

> *Research question 2: What factors are important to assign the highest score to the collection with the largest amount of relevant documents?*

Query logs contain queries submitted to a search system. The index of the search system may contain a lot of keys that are never queried for. Pruning the index using query logs removes unpopular keys from the index. This makes the index smaller, hopefully without affecting the retrieval performance for the majority of searches. These thoughts lead to the following question:

> *Research question 3: What is the effect of pruning highly discriminative key indices using query logs?*

Using links in web pages, Google PageRank can assign authority scores to web pages to enable ranking of those pages. This research focuses on collection selection, so the granularity of items to find is at collection level. It would be interesting to see if a link-based ranking algorithm works on collection level:

> *Research question 4: What is the effect of using distributed collection-level "PageRank" scores of peers to rank and select them?*

1.6 Thesis outline

The next chapter describes related work on distributed information retrieval that is relevant for this research. It describes several well-known collection selection methods, explains the concept of HDKs, introduces query-driven indexing techniques and describes distributed PageRank.

Chapter 3 describes the prototype that has been developed in order to answer the research questions. Chapter 4 explains the evaluation approach that was used to compare the collection selection performance of the prototype's different collection selection methods. Chapter 5 gives technical details about the index structure of the prototype that is capable of storing tens of millions of index entries.

Chapter 6 presents the evaluation results and interprets those results from various viewpoints. The thesis ends with a conclusion in Chapter 7 and a description of promising research directions in Chapter 8.

2

Literature about distributed information retrieval

Copy from one, it's plagiarism; copy from two, it's research.

– Wilson Mizner (1876–1933)

This chapter describes work related to distributed information retrieval in general, and collection selection in particular. The chapter starts by describing several well-known collection selection methods. It is followed by what can be called the foundation of this research: highly discriminative keys. As indexing with highly discriminative keys can result in large indices, the succeeding section discusses some approaches to reduce the index size using queries. The final section discusses distributed versions of the PageRank algorithm.

2.1 Collection selection

The essence of collection selection is selecting those collections that contain the documents that are relevant to the query of a user. Different methods can be employed to select such collections. This section describes four collection selection methods that are used by well-known collection selection systems.

2.1.1 Vector space model

The vector space model represents index items and user queries as weight vectors in a high dimensional Euclidean space [58]. Applied to the area of collection selection, the index items are document collections. The model uses the weight vectors to calculate the distance (or similarity) between document collections and queries. The collections with the smallest distance to the query are ranked highest.

GlOSS stands for Glossary-Of-Servers Server and is a system that uses statistical metadata about collections to create summaries [29]. These summaries can be used for collection selection. The system was extended by using the vector space model, resulting in the generalized Glossary-Of-Servers Server (GGLOSS) [28]. To measure the distance between a document collection and a query, gGlOSS needs two vectors for each collection c_i with:

1. The document frequencies $df_{i,j}$ for term t_j in collection c_i is the number of documents in c_i that contain t_j.
2. The sum of the weights of t_j of all documents in c_i: $w_{i,j}$. The weight of a term in a document is typically a function of:
   - the number of times that t_j occurs in a document, and;
   - the number of documents in c_i that contain t_j.

2.1.2 Cue validity

The *cue validity* can be seen as a the likeliness that given a cue, an object falls within a particular category [27]. CVV is a collection selection method that uses cue validity [85]. The cue validity $CV_{i,j}$ of query term t_j for collection c_i measures the extent that t_j discriminates c_i from the other collections. It compares the ratio of documents in c_i containing t_j to the ratios of documents in other collections containing t_j. The cue validity is calculated as follows:

$$CV_{i,j} = \frac{DF_{i,j}}{N_i} \Big/ \left(\frac{DF_{i,j}}{N_i} + \frac{\sum_{k \neq i}^{|C|} DF_{k,j}}{\sum_{k \neq i}^{|C|} N_k} \right)$$

$DF_{i,j}$ is the document frequency of t_j in c_i. N_i is the number of documents in c_i and $|C|$ is the total number of collections.

The cue validity variance CVV_j is the variance of cue validities for all collections with respect to t_j. A large variance in cue validities means that t_j is more suitable to differentiate collections. This notion is used to measure the *goodness* of a collection regarding a query q with n terms. The calculation only requires document frequencies and the total number of documents in the collections. The collection goodness is used for collection ranking. The goodness of c_i with respect to query q is calculated for all terms t_j where $j \in q$.

$$G_{i,q} = \sum_{j=1}^{n} CVV_j \cdot DF_{i,j}$$

The approach is criticized for its improper handling of rare query terms [64]. Rare query terms will have a low variance in cue validities, and those terms will likely have low document frequencies. Those query terms are in fact very useful for collection selection, but their effect on collection selection is mitigated by the goodness formula.

2.1.3 Inference networks

CORI [14] is a collection ranking algorithm for the INQUERY retrieval system [13], and uses an inference network to rank collections. Fig. 2.1 shows a simple document inference network in which the leaves d represent the document collections. The terms that occur in those collections are represented by representation

nodes r. Flowing along the arcs are probabilities that are based on document collection statistics. Opposed to TF.IDF, the probabilities are calculated using document frequencies *df* and inverse collection frequencies *icf* (DF.ICF). The inverse collection frequency is the number of collections that contain the term. An inference network with these properties is called a collection retrieval inference network (CORI net).

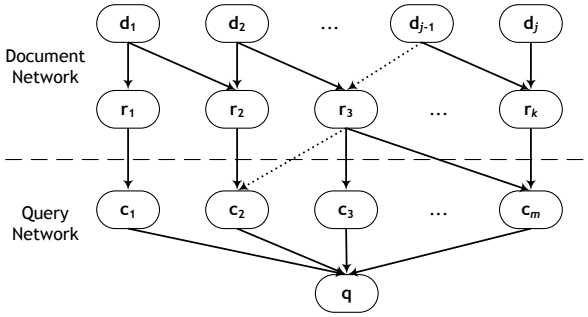

Fig. 2.1. A simple document retrieval inference network (figure from Callan *et al.* [14])

Given a query q, the network is used to obtain a ranked list of collections using a modified version of the INQUERY ranking algorithms. The query is forwarded to the top ranked collections, in order to obtain result lists that can be merged to get a ranked document list. Obtaining the ranked collection list is done by calculating the belief $p(r_k|c_i)$ in collection c_i due to the observation of query term r_k. The belief is calculated using Formula 2.4. Table 2.1 presents the parameter explanations. The collection ranking score of c_i for query q is the sum of all beliefs $p(r_k|c_i)$, where $r \in q$.

To improve retrieval, the component L is used to scale the document frequency in the calculation of T [13, 56]. L is made sensitive to the number of documents that contain term r_k (using b) and is made large using l. L should be large, because *df* is generally large. Tuning of these two parameters is required for every data set. According to D'Souza *et al.* [25], proper tuning is impossible because correct settings are highly sensitive to data set variations; the value of l should be varied largely even when keeping the data set constant while only varying the query type. The authors conclude that standard CORI is not justified as a collection selection benchmark.

$$L = l \cdot ((1-b) + b \cdot cw_i / \overline{cw}) \tag{2.1}$$

$$T = d_t + (1-d_t) \cdot \frac{\log(df)}{\log(df+L)} \tag{2.2}$$

$$I = \frac{\log\left(\frac{|C|+0.5}{cf}\right)}{\log|C|+1.0} \tag{2.3}$$

$$p(r_k|c_i) = d_b + (1-d_b) \cdot T \cdot I \tag{2.4}$$

b	constant that is varied between 0 and 1
l	constant that controls the magnitude of L
cw_i	number of words in c_i
$\overline{cw}$	mean number of words in the collections
$\|C\|$	number of collections
df	number of documents containing r_k
cf	number of collections containing r_k
d_t	minimum term frequency component when r_k occurs in c_i, default is 0.4
d_b	minimum belief component when r_k occurs in c_i, default is 0.4

Table 2.1. Parameters for belief computation in CORI

2.1.4 Language modeling

A language model can be used to assign a probability to the likeliness that a phrase has been generated using a modeled language. For example, there is a higher probability that "knowledge is of two kinds" was generated by an English language model as opposed to "of is kinds two knowledge". Language models were first used in speech processing, and were later used for Information Retrieval in 1998 [34, 52]. Language models have also been used for collection selection [67, 83]. One language model can be used to model one document collection. With such language models, probabilities can be calculated that collections generated a given query.

INDRI is part of the Lemur project[1] and is an improved version of the INQUERY retrieval system [77]. Compared to INQUERY, INDRI is capable of dealing with much larger collections, and allows for more complex queries due to new query constructs. The most important improvement is the formal probabilistic document representations that use language models instead of TF.IDF estimates. The combined model of inference networks and language modeling is capable of achieving more favorable retrieval results than INQUERY [47].

The different probabilistic document representation means that the term representation beliefs are calculated different compared to CORI (as described in Section 2.1.3):

[1] http://www.lemurproject.org/

$$P(r|D) = \frac{tf_{r,D} + \alpha_r}{|D| + \alpha_r + \beta_r}$$

The belief is calculated for representation concept r of document node D (in document collection C). Examples of representation concepts are a term in the body or title of a document. D and r are nodes in the belief network. The term frequency of representation node r in D is denoted by $tf_{r,D}$. α_r and β_r are smoothing parameters.

Smoothing is used to make the maximum likelihood estimations of a language model more accurate, which could have been less accurate due to data sparseness, because not every term occurs in a document [86]. Smoothing ensures that terms that are unseen in the document, are not assigned zero probability.

The default smoothing model for INDRI is Dirichlet smoothing, which affects the smoothing parameter choices. In setting the smoothing parameters, it was assumed that the likeliness of observing representation concept r is equal to the probability of observing it in collection C, given by $P(r|C)$. This means that $\alpha_r/(\alpha_r + \beta_r) = P(r|C)$. The following parameter values were chosen:

$$\alpha_r = \mu P(r|C) \tag{2.5}$$
$$\beta_r = \mu(1 - P(r|C)) \tag{2.6}$$

This results in the following term representation belief, where the free parameter μ has a default value of 2500:

$$P(r|D) = \frac{tf_{r,D} + \mu P(r|C)}{|D| + \mu}$$

A more detailed explanation can be found in the Indri Retrieval Model Overview [46]. Note that the formula described above is intended for document retrieval, in which case there is only one collection. It is possible to use INDRI for collection selection, by merging all documents of one collection to one single large document, and considering many of such large merged documents as the collection. The effectiveness of this use of INDRI has been demonstrated by several researchers [3, 63].

2.1.5 Discussion

Section 2.1.3 already described that it was not possible to set the parameters of CORI such that the retrieval results are consistently good. D'Souza *et al.* enforced their arguments by showing that HIGHSIM outperformed CORI in 15 of 21 cases [24]. Si and Callan [66] also found limitations with different collection sizes. Large collections are not often ranked high by CORI, although they often are the most promising collections.

Despite those issues of effectiveness, CORI remains an algorithm that has outperformed other algorithms like CVV and GGLOSS [20, 54].

Although INDRI is relatively new compared to the other discussed approaches, it shows favorable results in recent experiments [6, 16, 63, 84]. INDRI has a better formal probabilistic document representation than CORI and INDRI is an improved version of INQUERY. The INQUERY search engine forms the basis of the CORI collection retrieval model and CORI has outperformed many other algorithms. That is why we will use the language model as implemented by INDRI as our baseline collection selection system.

2.2 Highly discriminative keys

This section discusses a scalable indexing approach for document retrieval in a structured peer-to-peer network. In a structured peer-to-peer network it is possible to quickly reach a peer using efficient routing facilities.

2.2.1 Distributed hash table

A collection is a set of documents, where each document contains many terms. To provide fast search, an index can be implemented using a hash table. A hash table maps every key (term) to a *posting list*. A posting list contains the locations of the documents in which the associated term occurs.

Such an indexing structure can be distributed over many peers in a peer-to-peer network and is called a *distributed hash table* [55, 57, 76]. Every peer is responsible for a small number of keys and associated posting lists. To answer a multi-term query of a user, the peers are contacted that manage the requested keys. The corresponding posting lists are retrieved from the peers. The intersection of those posting lists gives the document locations that contain all query terms.

The *vocabulary* of a distributed hash table is defined by the different keys that are stored in the index. The vocabulary is relatively small if single terms are used, but the posting lists can be large due to the many term occurrences in the different documents. Whenever a query term is requested, the entire posting list will have to be transmitted, which is inefficient in a peer-to-peer network and does not scale to web search.

2.2.2 Indexing rare keys

Luu *et al.* proposed indexing of rare keys to reduce the storage and communication cost in a peer-to-peer network by increasing the vocabulary, but reducing the posting list size [44]. The rare keys can consist of one or multiple terms. The keys and associated posting lists are obtained as follows:

1. *Local vocabulary generation (selection of keys)*: possible keys are generated by making combinations of terms occurring in one document. To limit the

number of combinations, a proximity filter is applied that only allows combinations of terms appearing close to each other.
The main idea is that locally rare keys are selected, assuming that those keys will have a higher probability of being globally rare. Selected keys should not be redundant, which means that all subset keys must be frequent keys. The selected keys can not be globally frequent, so the peers are informed about this information.

2. *Local selection of postings for the keys in the vocabulary*: to keep the system scalable, the global posting list for a key is limited to DF_{max} documents. The peer will have to identify its DF_{max} documents with the highest key occurrences. Those documents are added to the posting list, which is used for the global selection of postings.
3. *Global selection of postings*: a posting list with at most DF_{max} postings is stored for every globally rare key. The stored postings have the highest number of key occurrences. A globally rare key is a key that occurs at a predefined maximum number of peers. A globally rare key is called a *highly discriminative key*, because such a key is very good in selecting a small number of peers that may contain relevant documents for a given query.

Later work on highly discriminative key indexing comprised improvements for retrieval performance and more efficient multi-term query answering [51]. It became possible to add locally frequent keys to the global index when those keys are highly representative for a document. Also, peers are able to store full posting lists for globally rare keys. Peers could also send a posting list with DF_{max} postings when a key was globally frequent, but locally rare.

As this significantly increases the index size, query-driven indexing was applied to reduce the vocabulary and posting lists were truncated to only store the postings with the highest key occurrences. The next section discusses query-driven indexing in more detail.

2.2.3 Discussion

Research results demonstrated that the retrieval performance is comparable to a state-of-the-art centralized relevance computation scheme (BM25) on three different test corpora, namely the TREC WT10g corpus, Wiki pages and Google cache results [51, 71]. Scalability analyses provided evidence of the plausibility that the approach would work in large scale peer-to-peer text retrieval system, because the index growth is bounded by a maximum posting list size [69, 72].

Tinselboer confirmed good scalability properties and tested the retrieval performance by measuring the top x overlap of the result lists of two systems [79]. Overlap measurement was also used in earlier evaluations of HDK based indexing [51, 69, 71]. Keeping all other variables the same, the first system used a centralized single term index with full posting lists and the second system used an index with rare keys with truncated posting lists. Based on low overlap ratios, he concluded that the retrieval performance was poor. We believe this conclusion is too premature, because the result list overlap only shows whether two

approaches produce comparable rankings. It is not even unlikely that the rare key indexing approach produced better document rankings, because it exploited query term proximity in documents. A good way to test such claims is by using relevance judgments of documents.

2.3 Query-driven indexing

In the process of creating HDKs, many keys will be created that are never queried for. By pruning those keys from the index, it is possible to reduce the index size. Care must be taken in removing keys, as retrieval performance can be reduced when too many keys are removed [22, 43].

2.3.1 Query-adaptive partial distributed hash table

Klemm *et al.* calculate whether it is worth to store a key in a distributed index in a peer to peer network [38]. The decision to store keys is based on a minimum query frequency of a key. The minimum query frequency is calculated using Equation 2.7 which depends on three variables:

cIndKey: The cost of storing the key in the index for a fixed period of time. Compared to a centralized index, the distributed index cost may be higher due to peers joining and leaving the network. This introduces costs such as routing table maintenance costs and update costs for inserting, overwriting and deleting keys at peers.

cSUnstr: The cost of searching a key in the (unstructured) network without the aid of an index by flooding the network with a query.

cSIndex: The cost of searching the key in the index.

$$fQry_k > \frac{cIndKey}{cSUnstr - cSIndex} \tag{2.7}$$

The minimum query frequency $fMin$ is set to the smallest number of $fQry_k$ for which Equation 2.7 still holds. Klemm *et al.* assume that the query frequency for all n unique keys follows a Zipf distribution with parameter α. When the keys are ordered by query frequency, every unique key has a rank r. Equation 2.8 gives the optimum number of keys that is worth indexing.

$$1 - \left(1 - \frac{r^{-\alpha}}{\sum_{x=1}^{n} x^{-\alpha}}\right)^{q_t} \geq fMin \tag{2.8}$$

In this equation, q_t is the total number of queries that are posed to the system per fixed period of time. The left hand side of the equation models the probability that a key is queried at least once per fixed period time. The rank r is chosen as high as possible such that the equation still holds. This gives r as the maximum number of keys worth storing in the index.

2.3.2 Pruning a broker index using a query log

Shokouhi *et al.* propose to prune a broker index by removing the keys that do not occur in a query log [65]. Their motivation is that terms not appearing in the query log can be removed from the index without affecting retrieval performance. Although this is true, it requires additional effort to deal with new unseen queries.

Also, they argue that – in a distributed information retrieval scenario with a broker index – query based pruning is more preferable than pruning based on global collection statistics. Pruning based on global collection statistics is deemed unreasonable because a term frequency can be low in one local collection, but high and critical for another collection.

However, at the expense of an acceptable decrease in retrieval performance, it is possible to reduce the global index size using collection statistics, by only maintaining the term posting list entries for the top x collections with the highest term frequencies. This has been demonstrated by Podnar *et al.* with HDKs [50]. Compared to a centralized system, they were able to achieve almost 94% overlap of the result lists by only storing the locations of the top 500 documents. Besides index size reduction, it also helps in limiting the number of transmitted postings (only 7.6% compared to the non-pruned centralized system).

Shokouhi *et al.* tested their pruning strategy in an uncooperative environment. Therefore, they took query-based samples of the collections. They removed all terms from the samples that did not occur in the query log and the remaining terms were used for indexing. Their results show a broker index size reduction of 22%–28% compared to an unpruned index, while maintaining equal retrieval performance.

2.3.3 Indexing based on query popularity

Skobeltsyn *et al.* extended HDK indexing by making the index generation dependent on the user queries [72]. They start by identifying three types of index items:

Basic index item: A single term key with its associated full posting list. The full posting list is stored at the peer responsible for that key.

Candidate index item: A multi-term key k with 2 to $s_{\max}$ terms, where $s_{\max}$ is the maximum number of terms in a HDK. The key has an associated query frequency counter. A candidate key is created when all smaller subkeys are already stored in the global index as basic or active index items.

Active index item: A multi-term key with a query frequency counter and a truncated posting list. A candidate key is activated when the query frequency counter exceeds a minimum query frequency (making it a popular key).

Global index construction starts with adding basic index items for all single terms in the document collection. The rest of the index construction is entirely driven by user queries; candidate index items are created as soon as they meet

the criteria for creation and candidate index items are activated as soon as the query frequency for the corresponding keys are high enough. A truncated posting list should be generated as soon as an item is activated. Peers are notified about this using the opportunistic notification mechanism [70]. This efficient broadcast method ensures that a peer receives a notification only once and forwards it to at most $\log n$ peers, where n is the number of peers in the network [21]. Notification messages are low priority messages that can be appended to high priority messages. Peers wait for high priority messages needing to be sent, or they send the low priority message when a time period expires.

Index updates are also driven by queries. Index items that become unpopular over time are deactivated. Postings in the index are removed when a document in a result set is inaccessible, thereby keeping the index up to date.

2.4 Distributed PageRank

In Information Retrieval, only finding the relevant documents is not sufficient; documents will have to be ranked in such a way that the most relevant documents appear on top of the result list. Several link-based algorithms have been developed for this goal, such as PageRank [10], HITS [37] and SALSA [42]. PageRank is the best known, so more research has been directed at using this algorithm in a distributed way. This section describes several of those algorithms.

2.4.1 Google PageRank

Google PageRank [10] is a static ranking algorithm, meaning that the rank of the documents is fixed, independent of the actual search terms. The general idea is that web pages contain links to other pages, thereby conferring importance to the linked pages. When important pages link to other pages, more importance is conferred. This gives an estimate of the relative importance of web pages.

This notion is captured in a model with a random surfer. A random surfer randomly clicks on links and visits a new page once he gets bored of clicking on links. When a page has many incoming links, the probability is higher that a random surfer ends up at that page. These concepts can be used to assign scores to pages. The scores, or PageRanks, can be precomputed for all pages and can be used to rank pages from a result set. The PageRank of page A is calculated using Formula 2.9.

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right) \tag{2.9}$$

In this formula, pages $T_1 \ldots T_n$ have links to page A and $PR(T)$ is the PageRank of page T linking to page A. $C(T)$ is defined as the number of links going out of page T. The formula uses a damping factor d, which models the probability that a random surfer keeps clicking on links, instead of visiting a random new page.

Power iteration is an algorithm to calculate the eigenvalue with the greatest absolute value of a matrix: the principal eigenvector. PageRank uses an adjacency matrix to store the link structure of the World Wide Web graph. The components of the principal eigenvector of such a matrix are the PageRanks. The power iteration converges relatively slow, making it interesting to distribute or optimize its calculation.

2.4.2 Distributed calculation

The link structure of the World Wide Web can be plotted with an adjacency matrix **A**, in which entry a_{ij} is 1 if page i links to page j and 0 if no such link exists. There are many billions of web pages, but pages only link to a small number of them. This results in a sparse link matrix. Web pages tend to link to multiple pages that reside on the same host. This results in a sparse link matrix with a block structure when the pages in the matrix are grouped by host [36]. Several distributed approaches exploit this sparsity and block structure in the link matrix to accelerate PageRank calculation:

- Local PageRanks scores can be calculated using the link structure of each host, assuming that links to pages outside the host do not exists. Kamvar *et al.* [36] calculate a BlockRank to estimate the relative importance of the hosts (or blocks) in the global link matrix. The BlockRank is used for weighting the local PageRanks. The weighted local PageRanks are used as starting PageRanks values for the regular iterative PageRank algorithm. This reduces PageRank computation by about a factor two due to faster convergence of the local PageRanks and BlockRanks, but it still requires centralized storage and computation of the global PageRank.
- Wang and De Witt have a comparable approach with ServerRanks that are used to refine local PageRanks [81]. As they do not compute the global PageRanks, the local PageRank scores are no true PageRank scores, but it reduces computation time, communication cost and storage cost. The "PageRank" scores are stored locally at the node in a distributed retrieval system, and are included with the result set when that node is queried. The scores can be used to merge and rank results from multiple result lists.
- Also starting with local PageRank computation, Zhu *et al.* use iterative aggregation-disaggregation with Block Jacobi smoothing for distributed global PageRank score computation [87]. It is an efficient way to compute ergodic probabilities of large Markov chains and can calculate PageRank scores about 5-7 times as fast as the traditional power iteration method.
- Bradley *et al.* use hypergraph partitioning to test two link matrix decompositions [9]. The first approach makes 1-dimensional matrix decompositions, which means that either rows or columns of the matrix are sent to processors to compute the vector products. The different processors cooperate to calculate the PageRanks. The second approach makes 2-dimensional matrix decompositions, which means that blocks of the matrix are assigned to a processor. The matrix decompositions are done in such way that interprocessor

communication is minimized while balancing the load on the different processors. Although their method has been developed for parallel PageRank computation, it would be interesting to see if it could be applied for distributed PageRank computation.

Sankaralingam *et al.* [59] proposed a method to distributedly calculate true PageRank scores. All documents on all peers start with an initial value. Next, the PageRanks are sent to out-linked peers that calculate updated PageRanks for their documents. Out-linked documents existing on the same peer are updated without additional network traffic. This procedure is asynchronously iterated until the PageRanks converged. Tests showed that 17 update messages per document were required for convergence in a test network with 40 peers and 2,000 documents. When the number of documents in the network was gradually doubled, 8.6 messages per document were required to converge the scores. This approach allows for continuous updated PageRank scores using incremental PageRank computation.

Overlap in collections may be an important factor in distributed information retrieval. Documents can be duplicated across the network to increase document availability in a peer to peer network. In our scenario, as described in Section 1.2, it is possible that search engines index documents from other sites. In both cases, the same documents can be found via multiple peers, which has consequences for the PageRank calculation. This can be solved by deduplicating documents in the network before starting PageRank calculation, or modifying the algorithm for PageRank calculation to deal with the effects of duplicates.

Parreira and Weikum developed the JXP algorithm that is capable of dealing with overlap to compute PageRank-like authority scores [48]. The algorithm starts by calculating PageRanks for the pages in the locally available web graph. The web graph uses a world node to model the unknown external nodes. The resulting local scores are called PageWeigths. Next, peers randomly meet with other peers. The two local web graphs are temporarily joined. Duplicate nodes are merged to one node, taking an average of the two PageWeigths. The PageRank algorithm is used to calculate new PageWeigths. PeerWeigths can optionally be used in the algorithm to compensate PageWeigths. This could be desirable for several reasons, such as a large difference in local index size or more knowledge about the rest of the world because a peers has had more meetings.

For collection selection, it is not required to have PageRank scores for the actual documents. It suffices to have scores for the collections that contain the relevant documents. Wu and Aberer introduced SiteRank which is an authority based scoring that works at the granularity of web sites [82]. Their research found evidence for a correlation between PageRank and SiteRank, making it reasonable to assume that many pages of important sites are important and a site is probably important when it has many important pages.

2.4.3 Discussion

Much research has been directed at parallel or distributed PageRank calculation. Many solutions decompose the problem of calculating the eigenvectors of the huge link matrix by splitting the matrix in smaller matrices. The smaller matrices are based on the links in documents that exist in one host, peer or site. Hosts, peers and sites can be seen as a collections of documents

In 2000, Savoy and Rasolofo tested four link-based retrieval systems on document collections [61]. The WT10g test corpus was split in four 2.7GB collections. Each of the four collections were queried and the results were merged by employing different merging strategies, such as raw score merging and CORI. The results showed that merging based on link information is only marginally useful, compared to other merging strategies.

Hawking and Thomas tested two link-based server selection methods (highest anchor ranked page and anchor weighted sum) on the TREC .GOV corpus which has 7792 servers [31]. The methods were compared to well-known selection methods such as CORI [14], Kullback-Leibler [67] and REDDE [66]. They concluded that link-based approach gave the best overall performance. This is interesting, because Kullback-Leibler is comparable to the baseline language modeling approach that was described in Section 2.1.4 [67].

Tsaparas tested several link-based algorithms and was surprised to find that simpler algorithms, such as InDegree – counting the number of incoming links, produced better rankings than more sophisticated algorithms such as PageRank [80]. He concluded that most link-based ranking algorithms favor the wrong parts in the link graph; the favored parts are not strongly correlated with the largest number of relevant documents in a collection. These findings were endorsed by Amento *et al.* [2], who found that ranking pages using the number of pages on a site works almost as good as link-based ranking.

Even though PageRank is not undisputed on document level, it remains interesting to see how higher level abstractions of PageRank perform in selecting collections with many relevant documents; especially when the number of collections is large.

2.5 Summary

This chapter started by describing several popular collection selection methods. Language modeling was introduced and the need for smoothing in those models was explained. Language modeling with Dirichlet smoothing will be used as the baseline collection selection method in this research.

Next, the chapter explained how HDKs were used for selection of relevant documents. This research will apply the same technique for collection selection, which will be explained in the next chapter. Three query-driven indexing approaches were described, of which the simplest approach by Shokouhi *et al.* will be used in our research. This simplest approach uses the query log as a filter for

interesting index items. It can be quickly implemented and can give a good idea about the obtainable index size reduction.

Finally, many distributed PageRank approaches exploit the block structure in the adjacency matrix of the World Wide Web link graph. The matrix contains blocks of links because many pages from one site link to many pages of a small number of other sites. This shows a link-based relationship between sites, which can be exploited to speed up PageRank computation. This suggests that a distributed link-based ranking algorithm can be made to rank collections instead of pages.

3

Sophos: a highly discriminative key prototype

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

– Franklin D. Roosevelt (1882–1945)

Sophos[1] is a prototype of a collection selection system that uses HDKs to index collections. The keys are used to answer queries by assigning scores to the collections to enable collection selection. Using a scoring function, the scores are used to generate a collection ranking for a given user query. Optionally, the prototype can use PageRank-like collection scores. A general overview is depicted in Figure 3.1.

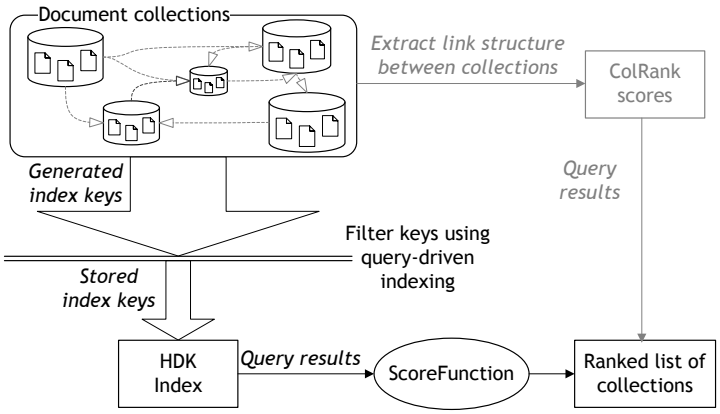

Fig. 3.1. General overview of indexing and collection selection system

This chapter starts by describing how the HDK index is created and then explains how the index size is kept manageable using a simple form of query-driven indexing. After that, ColRank – a PageRank-like collection ranking algorithm – is introduced. The chapter concludes with a detailed discussion about ScoreFunc-

[1] Sophos stands for "Sanders OnderzoeksPrototype: Hoogst Onderscheidende Sleutels" (which translates to Sanders Research Prototype: Highly Discriminative Keys). The ancient Greek word σοφος means wisdom or knowledge.

tion; a function that ranks the collections using the scores from the query results generated by querying the HDK index.

3.1 Highly discriminative keys

Building an index based on HDKs consists of two phases. The local indexing phase consists of building indices for each collection and is performed locally at every peer. In the global indexing phase, peers send a part of their local index to a broker. The broker constructs a global index from all local indices, which can be used to answer queries.

3.1.1 Local indexing

Local indexing starts with the generation of term sets statistics. First, single term statistics are generated by looking at the term frequencies of every word in the collection, without looking at document boundaries in the collection. A term set is called *frequent* when it occurs more times than a term set frequency maximum $tf_{\max}$. Every infrequent single term is added to the local index together with its frequency count. The frequent keys are stored for further analysis.

The next step in local index building is the generation of term sets consisting of two terms. The basis of every double term set is a frequent term. For every frequent term in the collection, frequency statistics are created for combinations of terms that occur in a particular window size *ws*. The window size denotes the number of words *after* a particular word with which combinations of term sets can be made with the first word. The concept is best explained by an example.

- We start with the following sentence: “The driver with the most gold medals at the end of the season becomes world champion.”
- When earlier analysis revealed ‘gold’ to be a frequent term, and the window size is set to 4, the following 4 term sets are created: “gold medals”, “gold at”, “gold the” and “gold end”. Term set frequency statistics are maintained for all generated sets.

This procedure is repeated for every frequent single term. The result is a list of double terms with corresponding frequency counts. Once again, the term set frequency maximum defines which term sets are frequent and will be used for further analysis, and which term sets are infrequent and will be stored in the local index. This procedure can be repeated up to the moment that the maximum window size has been reached or when a predefined maximum number of terms in a term set has been reached. This maximum is denoted by $h_{\max}$.

Summarizing, the local index consists of tuples with term sets and frequency counts.

3.1.2 Global indexing

The infrequent term sets from the local indices are sent to the broker. The broker stores the term set with an associated posting list in its global index. A posting list contains tuples with collection identifiers and corresponding term set frequencies.

When a term set is inserted into the global index, it is called a highly discriminative key. The global index will contain a maximum number of collections per HDK, denoted by the collection maximum *cm*. As soon as the maximum number of collections is reached for a particular HDK, the *cm* collections with the largest term set frequencies will be stored in the global index.

The global index consists of tuples with term sets and an associated set of collection identifier counters. A *collection identifier counter* is a combination of a collection identifier and a term set frequency counter. A collection identifier is a short representation of a collection where the term set occurs.

3.2 Query-driven indexing

A list of popular keys can be created by extracting all unique queries from a query log. The global index with HDKs can be reduced in size if keys are pruned that are not queried for. Every locally infrequent key that occurs in the unique query list will be inserted in the global index, the other keys are filtered out.

By pruning the index in this manner, we employ the strategy as described by Shokouhi *et al.* [65]. This is not the most desirable approach for query-driven indexing, because it is unable to deal with unseen query terms. However, it will give a good idea about the possible index size reduction and the loss of retrieval performance.

3.3 ScoreFunction: ranking the HDK query results

3.3.1 Identifying query result parameters

Once the global index has been built, the system is ready to be queried. The global index contains HDKs with h terms, where h varies from 1 to $h_{\max}$. In Sophos, $h_{\max}$ is set to 3. Every key has an associated posting list, which contains tuples of collection identifiers and counters. A counter indicates the number of occurrences of a certain key within the corresponding collection. The value of that counter can never be higher than the term set frequency maximum, $tf_{\max}$, as a key would otherwise have been locally frequent and new HDKs would be generated when the maximum number of terms, $h_{\max}$, was not yet reached.

When a user poses a query with q terms, e.g. abcde with $q = 5$, the query is first decomposed in query term combinations with length $h_{\max}$ (i.e. abc, abd, ..., bde, cde). This results in $\binom{q}{h}$ combinations. Each combination is looked up

in the global index. The results of each of those smaller queries are merged by summing the number of occurrences per collection. The sum of all counters, c, has a potential maximum of $\binom{q}{h} \cdot tf_{\max}$. It may happen that this procedure results in little or no collections where an HDK occurs. The procedure is then repeated for smaller term sets; first term sets of two terms will be looked up in the index. When even this gives too few results, single terms will be looked up in the index. In the case that one collection contains two different combinations, e.g. both `abc` and `bce` occur in collection `X`, the number of occurrences are summed (this is c that was just introduced). However, it also interesting to note that 4 out of 5 query terms can be found in collection `X`. The number of query terms that can be found using HDKs of a particular length is indicated by n.

The query procedure as described above introduces a number of different parameters that can be relevant for result ordering. For convenience, the different parameters are displayed in Table 3.1.

c	the sum of term set occurrences within one collection, where every term set has an equal value of h
h	number of terms in an HDK
$h_{\max}$	maximum number of terms of which an HDK can consist
q	length of the query
n	number of distinct terms from a query term set found in a collection
$tf_{\max}$	maximum frequency of a term set in a collection

Table 3.1. Parameter definitions for query result handling

3.3.2 Desired ranking properties

To produce a desirable ranking, the query scores will have to be calculated in such a way that they enforce the desired ranking. Collection selection using HDKs should provide hints of good collections to forward a query to. The system is not built for exhaustive search giving the best possible answers. This is inherently impossible using HDKs, as whenever a term set appears too frequently in a collection, it is not included in the index. This phenomenon has consequences for the desired ranking and thus for the required score formula. In order of importance, we define the desired result ranking factors as follows:

1. If a longer HDK occurs in one or more collections, those collections should be ranked higher than collections that only have smaller HDKs appearing in them, e.g. collections that are found after querying `abc` ($h = 3$) should be ranked higher than collections that can only be found after querying `ab` and `bc` ($h = 2$).
2. More distinct query terms within a collection should produce a higher collection rank than a collection with less distinct query terms. So if the query

is abcde, collection X contains abcd ($n = 4$), and collection Y contains cde ($n = 3$), then collection X should be ranked higher than collection Y.
3. More occurrences of query term sets within a certain collection should result in a higher collection ranking.

An important detail to mention about querying and ranking is that collection X can be found after looking for HDKs with length h. When the same collection X is found after looking for HDKs with length $h - 1$, those results are discarded as the collection was already found using larger HDKs: the collection count remains unchanged. Counts c are only compared with other counts that are retrieved after looking for HDKs of the same length. The motivation for this behavior is that smaller HDKs tend to have higher counts. The latter is easily explained by the fact that a smaller term set has a higher chance of occurring in a certain window size than a larger term set. Therefore, it is unfair to compare counts obtained by looking for HDKs with different lengths.

3.3.3 ScoreFunction: the formula

To ensure the desired ranking, the scores will have to be generated in such a way that the most desirable property has the most influence on the score. In fact, the influence that the second desirable property should have, should never be bigger than the smallest increment in score that the first desirable property is able to make. The same holds for the second and third property. ScoreFunction has been devised to calculate a score s for a query result that ensures the desired behavior:

$$s = \log_{10}\left(\left[h - 1 + \frac{n-1}{q} + \frac{\sqrt{\frac{c}{(h_{\max}+1-h)\cdot\binom{n}{h}\cdot tf_{\max}}\cdot\alpha^{(q-n)}}}{q}\right]/h_{\max}\right) \tag{3.1}$$

The formula can be broken down into three components – each responsible for one desirable property. The three components are between the square brackets and will be described below. The logarithm in the formula ensures that the scores can be combined with other score formulas, such as ColRank. The remainder of the section discusses component score properties.

Property 1: Higher ranking with longer HDKs

The first component is $h - 1$. This component will have a higher value when longer HDKs are used to find collections. The minimum component value is 0 (as the minimum value of h is 1). The maximum component value is $h_{\max} - 1$.

Property 2: Higher ranking with more query terms in collection

The second property is expressed as $(n - 1)/q$. The component value will be higher when more query terms are found, so when n is higher. As n will be equal

to or smaller than q, the component value will never be higher than $\frac{q-1}{q} = 1 - \frac{1}{q}$. The minimum value is 0, because the minimum values of n and q are 1.

Property 3: Higher ranking with more term set occurrences

The final component is larger and needs a more detailed explanation, so it will be explained in three steps.

$$\frac{\sqrt{\frac{c}{(h_{\max}+1-h)\cdot\binom{n}{h}\cdot tf_{\max}} \cdot \alpha^{(q-n)}}}{q}$$

- Starting with $c/\left(h_{\max}+1-h\right)$, this part corrects the sum of term set occurrences within one collection (denoted by c). This is required, because an index can contain multiple keys that contain a particular query term.
  The keys in the index are ordered by term. For example, an index can contain the following keys: `efg`, `ae`, `be`, `cef` and `bce`. This example shows that the query term `e` can occur multiple times and at different positions, because a key can consist of up to h_{max} terms. A single term can occur at any of the $h_{\max}$ positions in a key. This means that the sum of scores can be $h_{\max}$ times higher than a query term set with $h_{\max}$ terms. A key with $h_{\max}$ terms can occur only once in the index, due to the ordering of the terms in the index; only one count c can be found per collection when $h = h_{\max}$.
- The maximum value of c (after application of the correction that was just described) is $\binom{n}{h} \cdot tf_{\max}$. By dividing the count with this number, the fraction will never exceed 1.
- The last part, $\alpha^{(q-n)}$, is used to lower the scores when not all query terms have been found. It ensures that good scores (with high values of c) become relatively more important when more query terms have been found in a collection. This increases the gap between good collections (with many relevant documents) and mediocre collections in favor of the good collections. The value of α should be between 0 and 1; 0.1 seems to be a reasonable value as this still keeps differences in counts noticeable in the final scores. Smaller values, e.g. 0.001 produce final scores that make it hard to see differences in query result scores when h and n are equal.

Finally, the square root is taken over the resulting score. Its purpose is to increase the sensitivity within the scores for lower counts, e.g. the difference in c of 1 and 10 will be more noticeable in the score than a difference between 291 and 300.

Effect of three properties on the scores

The sum of the maximum values of the property components is listed in Table 3.2. The sum of the values is $h_{\max}$. Formula 3.1 divides the sum of the component values by the same value, so the maximum score s is $\log_{10}(h_{\max}/h_{\max}) = -1$. The score will be more negative when at least one of the component values will be lower, ensuring a correctly ordered collection ranking.

	Maximum component value
Component for property 1	$h_{\max} - 1$
Component for property 2	$1 - (1/q)$
Component for property 3	$1/q$
Sum of maximum component values	$h_{\max}$

Table 3.2. Minimum and maximum values of the three formula components

3.3.4 Assigning weights to formula components

The previous section described ScoreFunction; a formula with three components representing three properties. It is the standard scoring formula and will be used in most Sophos tests.

Another formula was created by modification of the standard formula to experimentally derive which component should have the highest contribution in the score to rank the collections with many relevant documents on top of the result list. Using a β- and γ-parameter, the component score weights can be shifted. The component weights are assigned as follows:

Property 1: $(1-\beta-\gamma)\cdot \text{component}_1$
Property 2: $\beta \cdot \text{component}_2$
Property 3: $\gamma \cdot \text{component}_3$

This results in a modified *weight shifting ScoreFunction*:

$$s = \log_{10}\left(\frac{(1-\beta-\gamma)\cdot(h-1) + \beta\cdot\frac{n-1}{q} + \gamma\cdot\frac{\sqrt{\frac{c}{(h_{\max}+1-h)\cdot\binom{n}{h}\cdot tf_{\max}}\cdot\alpha^{(q-n)}}}{q}}{h_{\max}}\right) \tag{3.2}$$

A general idea about the importance of the different components can be obtained by testing collection selection ranking with different values of β and γ. We will test all different parameter combinations of $0 \leq \beta < 1$ and $0 \leq \gamma < 1$ with 0.05 step increments of the parameters.

3.4 ColRank

In order to answer the fourth research question about the effect of distributed PageRank scores for collections, the PageRank algorithm was modified to work on collection level. Section 2.4 described that it is possible to distribute PageRank calculation over peers. The section also described that the SiteRank algorithm – PageRank on web site level – can be distributed. This supports the notion that a PageRank-like algorithm on collection level can be distributed as well, so we focused on a centralized algorithm that will be faster to implement and less prone to errors.

The PageRank algorithm was introduced in Section 2.4.1 and is repeated for easy reference:

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right)$$

To make a centralized collection ranking algorithm, pages can be substituted for collections. Inspired by the original PageRank algorithm, we define our collection ranking formula, creatively named ColRank, as follows:

$$C(A) = (1-d) \cdot \frac{1}{\#\text{ sites}} + d \sum_{j=1}^{\#\text{ sites}} \frac{C_{j,\text{old}}}{NA(C_j)} \tag{3.3}$$

This formula calculates the ColRank for collection A. It uses a damping factor d and it uses the old ColRank scores of all collections. The old ColRank score of the j^{th} collection is denoted by $C_{j,\text{old}}$. The number of links from the j^{th} collection to collection A is denoted by $NA(C_j)$. The ColRank formula looks a bit more complicated, but the only real difference with the PageRank formula is the fraction with the number of sites. This ensures that all ColRank scores together sum up to 1. This is not the case with the PageRank formula: the PageRank scores sum up to 1 per link-connected graph of pages. The effect of adding this fraction is that unconnected collections or collections in small connected collection graphs do not receive relatively high scores.

This formula is used iterative over all collections, until the ColRank scores stabilize. Stabilization of scores is detected by looking at the differences. After calculating the new ColRank scores for every collection, two scores are known per collection: the new $C(A)$ and the old $C_{j,\text{old}}$, where A is the j^{th} collection. The Euclidean distance can be calculated between both scores and this distance can be calculated for every collection. When the sum of those distances is smaller than a threshold ϵ, the scores have stabilized and the calculation is stopped.

We will use the same value for the damping factor that was used in the original Google PageRank algorithm: 0.85 [10]. We will use a threshold ϵ of $1e-10$.

3.4.1 ColRank with highly discriminative keys

ColRank is a static ranking algorithm, so each collection will have the same score for every query. To make ColRank query-dependent, its scores can be combined with scores from another collection selection method.

To combine the ColRank scores with the scores from HDKs with query-driven indexing, both query result sets will have to be merged. Each result set contains top 1,000 results, when there were that many query results. The result lists do not need to have the same collections. When a collection occurs in both result sets, the average of both corresponding scores is calculated. Otherwise, the score counts for 100%.

It is unfair to count a collection score for 100% when it appears in only one result list, because it 'punishes' other collections for appearing in both result lists;

their good score is lowered by a worse score. It is not straightforward to find a substitute score for the missing result. It can not be replaced by 0, because that is the score of an infinitely good collection (due to the logarithm in the score calculation formulas). It cannot be replaced by the lowest score in the result list, because the scores can vary much per query. It can even be that the missing collection score appeared to be extremely low when more than 1,000 query results were generated. The variance per query and the possibly large variance between scores also make it hard to use a fixed minimum score. More research is required to find a more fair score combination.

3.5 Summary

Sophos has two approaches to rank collections: ColRank and several HDK based methods. ColRank is a PageRank-like query-independent collection ranking method. In the HDK based methods, every peer starts by building its local index. It does so by generating a large amount of term sets. A term set in a collection is sent to a broker when it meets the following conditions:

1. The term set has a collection frequency lower than $tf_{\max}$, and;
2. when query-driven indexing is enabled: the terms of the term set are a subset of the terms of any query in the query log.

Whenever such a term set is globally rare – so when it occurs in less than *cm* collections *or* when it has the top-*cm* highest term set frequency – it is called a highly discriminative key. Sophos also has a scoring algorithm (ScoreFunction) that ranks collections according to three properties. In order of importance, collections are ranked higher when:

1. collections contain longer HDKs.
2. more query terms can be found in a collection.
3. HDK occurrences have higher frequencies.

4

Evaluation of collection selection methods

Success has always been easy to measure. It is the distance between one's origins and one's final achievement.

– Michael Korda

This chapter describes the research approach and starts with a description of the data collections that will be used for the experiments. It describes the document corpus, the corpus splits and the accompanying test queries. Three different query logs will be covered as well. The chapter concludes with an explanation of the settings used for the different collection selection methods and a description of the evaluation procedure.

4.1 Data collections

4.1.1 WT10G corpus

The Web Track 10GB corpus (WT10g) was developed for the Text REtrieval Conference.[1] It is a subset of the 100GB Very Large Corpus, which is a subset of a 320GB Internet Archive crawl from 1997. The WT10g corpus consists of 10GB of processed web pages. The processing consisted of removing non-English and binary data documents. A large amount of redundant or duplicate data has been removed as well. Compared to regular TREC corpora, this web corpus should be more suited for distributed information retrieval experiments, due to the existence of hyperlinks, differences in topics, variation in quality and presence of duplicates [5, 19]. According to Ian Soboroff, WT10G is a good reflection of the web [73].

Some statistics on the WT10G test corpus were obtained from the web page describing the corpus[2]:

- 1,692,096 documents on 11,680 servers
- average of 144 documents per server with a minimum of 5 documents per server
- 171,740 inter-server links (within the corpus)

[1] http://trec.nist.gov/

[2] http://ir.dcs.gla.ac.uk/test_collections/wt10g.html

- 9,977 servers with inter-server in-links (within the corpus)
- 8,999 servers with inter-server out-links (within the corpus)
- 1,295,841 documents with out-links (within the corpus)
- 1,532,012 documents with in-links (within the corpus)

4.1.2 Splitting WT10G

To answer the first research question about the effect of document corpora on the collection selection, five splits of the WT10G document corpus were used for evaluation. Every split is a set of collections, every collection is a set of documents. The list below describes how the corpus splits were created. The names between parentheses indicates how the corpus splits are called in the rest of this thesis. The numbers 100 and 11512 indicate the amount of collections in the corpus split.

Split per IP (IP Split): Documents are put in collections based on the IP addresses of the site where a document was residing. This results in 11,512 collections, which is a little less than the 11,680 servers in the corpus. This means that multiple servers run at one IP address.

Split using clusters of peers (IP Merge 100): A cluster is created by grouping up to 116 collections, which results in 100 collections. Grouping is done in order of IP address. This corpus split simulates the scenario of indexing the search engines that index many servers.

Random splits (Random 100 and Random 11512): Two random splits are created: a split with 100 collections and another one with 11,512 collections. Every document is randomly assigned to one collection. The number of 11,512 collections was chosen to be able to compare a random split with the IP Split.

Split with overlap (Overlap3 11512): The split with overlap is an extension to the random split; every document is randomly assigned to three of the 11,512 collections.

Fig. 4.1 and Fig. 4.2 show the document distribution for the different splits. In the graphs, the collections of each split are ordered on amount of documents in the collections. The number of documents is plotted logarithmic due to larger differences between and within splits. Fig. 4.3 and Fig. 4.4 depict the size distribution in bytes of the collections in each split. In these graphs, the collections are ordered on collection size. The size is plotted logarithmic.

The graphs show that the collections of the random splits do not vary much in number of documents and size. This can be attributed to the uniform probability distribution of documents being assigned to collections. Fig. 4.2 and Fig. 4.4 show that the IP Merge 100 corpus split shows a strong drop in both size and number of documents for the smallest collection. This smallest collection is the 100^{th} collection within the split, which contains documents of the last 88 collections from the original IP split (in contrast to the 116 collections for every other clustered collection).

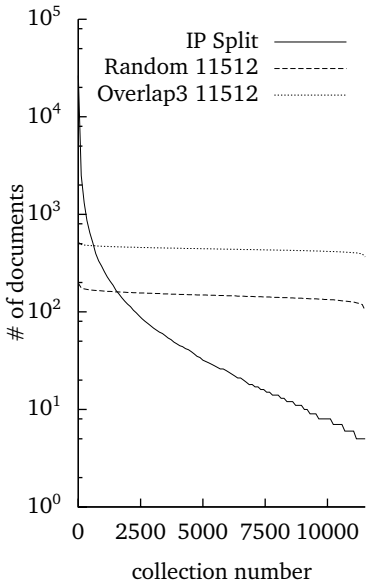

Fig. 4.1. Document distribution for splits with 11,512 collections

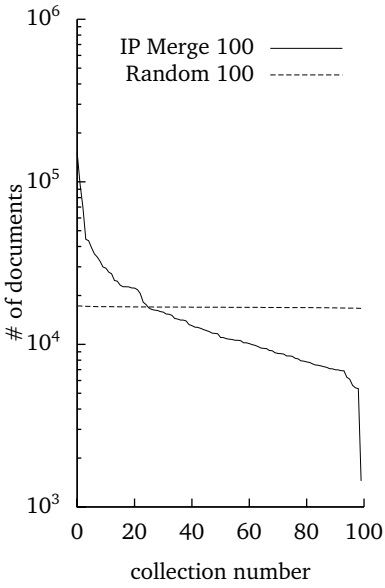

Fig. 4.2. Document distribution for splits with 100 collections

Despite this drop, there remains a large difference in number of documents and sizes of the collections. This is typical for the Internet, as there exist both very large and very small sites; the number of documents per server follows a Zipf distribution on the Internet [1]. The IP based splits show signs of conformance to a Zipf distribution, especially for the IP Merge 100 split (as shown in Appendix A). This means that the IP based splits can be compared to the Internet in terms of distribution of the number of documents and the sizes of the collections.

The *merit* of a collection is the number of relevant documents in a collection for a particular query. Appendix B: *Merit in corpus splits* shows that IP based corpus splits have a larger deviation in merit among the collections. This contrasts with random splits, which by approximation have the same amount of merit for each collection within the splits.

4.1.3 WT10g retrieval tasks

Broder identified three types of user intent in queries on the Internet: navigational, transactional and informational [11]. Navigational queries are about finding home pages. Transactional queries are concerned with buying things and informational queries are about finding information about topics.

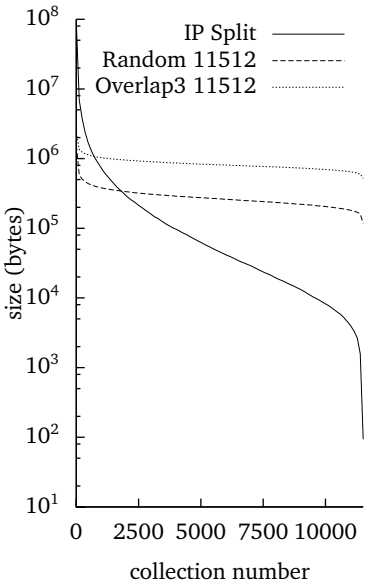

Fig. 4.3. Collection size distribution in bytes for splits with 11,512 collections

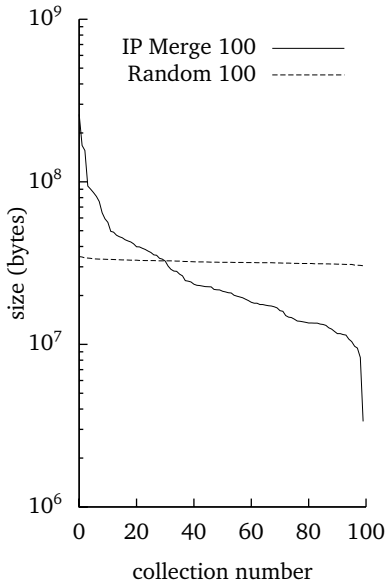

Fig. 4.4. Collection size distribution in bytes for splits with 100 collections

The WT10g test collection is accompanied with query sets, of which a selection was used for our evaluation. The selected query sets are divided in two types of queries:

- 145 navigational 'homepage-finding' queries: query numbers 1–145.
- 50 informational 'ad-hoc' queries: query numbers 501–550.

The navigational queries in the query set have a query number and a query text. The informational queries in the query set have a query number, a title, a description and a narrative description of the result that is considered relevant. The title is a small set of words which was used as the query text. The narrative descriptions of relevant results were used by humans to assign relevance judgments to documents for each query. The relevance judgments can be used to count the number of relevant documents in the collections, which in turn can be used to measure collection selection effectiveness.

The relevance judgments for the navigational query set classifies documents as either not relevant (indicated with a 0) or relevant (indicated with a 1). The informational query set has three types of relevance judgments: not relevant (0), relevant (1) or authoritative (2). There can be multiple authoritative documents in the document corpus for each main query, but some queries have no authoritative documents. Authoritative documents are always relevant documents, but

not vice versa. All authoritative judgments are converted to 1, so documents are either relevant or not relevant. This allows for evaluation of the collected amount of relevant documents.

4.1.4 AOL query log

AOL published a large query log, giving researchers the opportunity to do information retrieval experiments with a large data set. The AOL query log data consists of 21,011,340 queries from 657,426 users [49]. The queries were issued in a time span of three months from March 1, 2006 until May 31, 2006. The collection has been anonymized and consists of the following data fields:

AnonID: The anonymous user ID number.
Query: The issued query, converted to lower case and with removal of most punctuation.
QueryTime: The query submission date and time.
ItemRank: The rank of the clicked result (if the user clicked on a result).
ClickURL: The domain of the clicked result.

Although every user was assigned a unique ID, it remained possible to create user profiles about them. It has even been proven possible to link an ID to a real person. The release of the anonymized data set led to dissatisfaction on the Internet, forcing AOL to take it down on June 8, 2006.[3] In an ethical discussion about the use of query logs, Bar-Ilan states that academic researchers are eager to use the query logs, but hesitate to do so because of concerns regarding the privacy of the users [7]. We used a stripped version of the query log that only contains the actual queries issued (and none of the other metadata).

4.1.5 Excite query logs

Excite[4] has also published query logs. We used two Excite query logs. The first query log contains queries that were issued on September 16, 1997. It contains 1,025,907 queries of 211,063 users. Besides the actual queries, the log contains a time stamp and an anonymous user code per query. An extensive analysis of this query log is provided by Spink *et al* [75]. The second query log contains queries from 1999 and has 1,777,251 queries by 325,711 users. This log contains the same metadata per query as the Excite 1997 query log, but it also shows the number of additional result pages that were viewed by the Excite users.

Only the actual queries were used from both query logs; all additional information is discarded.

[3] More information on the AOL Search Query Log Release Incident can be found on http://sifaka.cs.uiuc.edu/xshen/aol_querylog.html
[4] http://www.excite.com/

4.2 Evaluations of collection selection methods

This section describes the approach followed for the evaluation of the different collection selection methods. First, a description of the general test procedure is given. This is succeeded by a description on the collection selection method specific settings. The evaluations were performed with every corpus split that was described in Section 4.1.2. The collections in the corpus split are used to simulate the contents at peers. Therefore, 'peers' and 'collections' can be used interchangeably.

4.2.1 Test procedure

Obtaining query results

The first evaluation step is calculating the merit for each of the 195 queries. A collection ranking can be created by descendingly ordering them on merit. This ranking is the best possible ranking and is the ranking that retrieval algorithms should produce.

The next evaluation step consists of indexing the collections and is covered in more detail in the next section. Once the index has been created, the retrieval system is ready to answer the queries. A total of 1,000 ranked query results (if available) is stored per query and used for further processing. The processing consists of adding merit values to the ranked query result lists to enable performance evaluation of the collection selection.

Traditional precision and recall

Most of the time, information retrieval works at document level. Metrics have been created to measure the retrieval quality of document selection systems. The best known metrics are precision and recall. Precision is defined as the ratio of relevant documents to all selected documents. Recall is defined as the ratio of selected relevant documents to all relevant documents in the corpus. Selecting all documents will give maximum recall, but results in a low precision. Selecting a small amount of relevant documents – only when the retrieval system is convinced they are relevant – gives a high precision, but a low recall if there are many more relevant documents. Therefore, obtaining a high precision and recall is usually a trade-off.

Our research aims at collection selection. One collection can contain many relevant documents. The performance metrics will have to be adapted for this scenario. This will be covered in the remainder of this section.

Performance metrics for distributed collection selection

After processing the query results, the *retrieval system ranking* (S) and *best ranking* (B) are known and statistics about them can be calculated. Inspired by metrics

from Gravano *et al.* [28], French *et al.* [26] use metrics to evaluate the performance of collection selection. The same metrics will be applied here, as they are a good approximation of the traditional precision and recall metrics. Before describing the metrics in formulas, several variables will be introduced.

Given query q and collection c_i, the merit within a collection can be expressed using $merit(q, c_i)$. The merit of the i^{th} ranked collection in rankings S and B is given by $S_i = merit(q, c_{s_i})$ and $B_i = merit(q, c_{b_i})$. Using these two variables, the performance of retrieval systems can be compared to the best possible ranking.

The obtained recall after selecting n collections can be calculated by dividing the merit selected by the retrieval system with the merit that could have been selected in best case. Gravano *et al.* defined the following metric for this purpose:

$$R_n = \frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} B_i} \tag{4.1}$$

This metric was slightly modified by French *et al.* to:

$$\widehat{R_n} = \frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n^*} B_i} \tag{4.2}$$

$$\text{where } n^* = \max k \text{ such that } B_k \neq 0$$

In this modification the denominator is changed to the *constant* total merit of all collections. Therefore, $\widehat{R_n}$ is the fraction of the total merit that has been accumulated by the top n selected collections, making $\widehat{R_n}$ comparable to the traditional recall metric. The difference between R_n and $\widehat{R_n}$ is that R_n can be 1 from the beginning (when the retrieval system results are equal to best possible ranking), whereas $\widehat{R_n}$ gradually increases to 1 (at which all relevant collections have been selected). We will call Formula 4.1 with its increasing denominator *cumulative recall*. Formula 4.2 with the total merit within the collection in the denominator will be called *total recall*.

A metric for precision was also defined by Gravano *et al*. P_n gives the fraction of top n collections that have non-zero merit:

$$P_n = \frac{\left|\left\{\text{sc} \in \text{Top}_n(S) | \text{merit}(q, \text{sc}) > 0\right\}\right|}{\left|\text{Top}_n(S)\right|} \tag{4.3}$$

In this formula, the counter of the fraction counts the number of selected collections sc that have a merit higher than 0 and occur in the top n of the ranked result list S ($\text{Top}_n(S)$).

4.2.2 Collection selection method specific settings

Now that the evaluation metrics have been explained, the remainder of this chapter describes the collection selection systems specific settings. It will begin with language modeling with Dirichlet smoothing, which was used for baseline evaluation. The description will then continue with explaining the settings for the custom built collection selection systems.

Language modeling with Dirichlet smoothing: the baseline

The INDRI search engine was described in Section 2.1.4. The section described that its default implementation of language modeling with Dirichlet smoothing can be used to model the large merged documents (collections). Using these models, the system can serve as a centralized search engine that has knowledge about all the terms in the corpus. The section also described that this approach has been used before with good results, so we will use this method as our baseline collection selection method.

According to Donald Metzler [46], the recommended smoothing type for INDRI is Dirichlet, which has one free parameter μ. We will use the default parameter value of $\mu = 2500$, which appears to be a good value [18].

For indexing the collections, we will use Krovetz word stemming which is provided as part of the Lemur toolkit. Stemming is a technique to reduce different words to common roots, e.g. 'doing' is converted to 'do'. It is a way to increase recall, especially when queries and documents are short as this is the hardest situation to match query terms to document terms [35, 39]. The combination of Krovetz stemming with our settings of INDRI is no uncommon approach [17, 18, 23].

In the rest of this thesis, this approach is referred to with the abbreviation LMDS (Language Modeling with Dirichlet Smoothing).

Sophos

The remainder of this section describes the evaluation specific decisions that were made to evaluate the retrieval performance of Sophos. The section starts with the collection and query preprocessing, continues with HDK indexing parameters and concludes with query driven indexing.

Collection and query preprocessing

Before local indexing begins, stop words are removed from the collection. Stop words are words that frequently occur in natural language text, such as 'the', 'and' and 'by'. The frequently occurring stop words make up almost 50% of the number of words in the WT10g corpus. Removing the stop words has the advantage that the corpus size is reduced. It also has two other advantages regarding term set generation:

1. Stop words are frequently occurring words. When those words are not removed from the corpus, many term sets would be made with those words. For example, the word "the" occurs on many places in a document. A term set would be generated for every infrequent word that is near "the", just to find term sets that are infrequent and do contain the frequent word "the".
2. As stop words make up almost 50% of the words in the corpus, we can effectively half the window size to get the same amount of combinations with more intrinsic words. This speeds up term set generation and drastically reduces the number of combinations that will be generated.

The disadvantage is that certain queries can not be answered anymore. A good example of such a query is finding information about the English pop group "The The".

Besides stop word removal, all words are converted to lower case and punctuation is removed from the collections. The collection will also be subjected to a word stemming algorithm. We used the Porter stemmer [53], because the official code is readily available on the Internet[5], in contrast to the Krovetz stemmer that is used for the baseline. The use of official code ensures that no errors are introduced by improper word stemming.

The downside of choosing another word stemmer is the possible introduction of differences in retrieval performance. Porter is a more aggressive stemming algorithm, which means that more words are stemmed to the same word stem. This could lower precision and recall for Sophos compared to the baseline. We think that the difference will be small, due to the nature of HDKs. The possible negative effect of a more aggressive stemmer is an increase of term set frequencies for particular term sets; if the effect is noticeable at all, it could be countered by increasing the term set frequency maximum $tf_{\max}$.

The same collection preprocessing steps are applied to the queries. It consists of stop word removal, converting everything to lower case, punctuation removal and word stemming. The processed queries are used for querying to test recall and precision of Sophos. An important part of querying is the ScoreFunction that was described in the previous chapter. The modified formula that was described in Section 3.3.4, will be used to test which element of the scoring formula is the most important to obtain the best ranking for a given query.

Highly discriminative key parameters

Section 3.1 introduced three local index parameters: the maximum key length $h_{\max}$, the maximum key frequency before calling a key frequent ($tf_{\max}$) and the window size ws. The only global indexing parameter was the collection maximum cm.

The test values for $tf_{\max}$ and window size ws are based on the numbers that were used in the HDK study of Luu *et al.* [44]. We use the following settings: $tf_{\max} = \{250, 500, 750\}$ and $ws = \{6, 12, 18\}$.

Several researchers measured the average query length on the Internet and found it to be 2.3 [40, 68]. We use these observations to set $h_{\max}$ to 3. Setting it smaller would require many intersections of term sets (with associated collections) to answer queries. Setting it larger would result in many term sets that are rarely queried for.

The tested global indexing parameter values for cm are 5, 10, 20 and 50.

[5] The official home page for Porter stemming algorithms can be found at: http://tartarus.org/~martin/PorterStemmer/index.html.

Query driven indexing

To answer the third research question about the effects of pruning the global index, three query logs will be used to filter keys: the AOL query log, the Excite 1997 query log and the Excite 1999 query log. A fourth query log will be created by removing all queries from the AOL query log that were issued only once. This fourth query log will also be used for filtering.

Besides looking at the actual index size reduction, the collection selection performance will be monitored. The latter is done by looking at precision and recall for the navigational and informational queries.

4.3 Summary

This chapter introduced modified versions of the precision and recall metrics to measure collection selection performance. The two formulas will measure the performance of five methods:

1. Language modeling with Dirichlet smoothing (the baseline)
2. Collection selection using HDKs
3. Collection selection using HDKs with query-driven indexing
4. Collection selection using HDKs with query-driven indexing and ColRank
5. ColRank

The metrics are used to measure the performance of the five methods on five different corpus splits, where the corpus splits have 100 or 11512 collections and where the collection composition is based on random document assignment or IP based document assignment. The chapter also described several query logs of which we will solely use the actual queries for pruning indices.

5

Sophos index implementation

Science is organized knowledge. Wisdom is organized life.

– Immanuel Kant (1724–1804)

This technical section is intended for developers that like to gain insight in a possible approach to efficiently store keys and associated information. As the amount of generated keys can become very large, a special structure is required to efficiently store keys, collection identifiers and counters. The chapter describes how every term of a key can be stored in 3 bytes. Next, the multi-level term index is described which is the base of local and global indices.

5.1 Unique term index

5.1.1 Corpus term analysis

The WT10g corpus contains 493,993,716 terms, of which 2,421,319 are unique terms. As the corpus is a crawl of the web, it contains a small percentage of insensible terms. Examples are character repetitions (like hundreds a's) or a large amount of concatenations of one of the most popular web queries. Because such terms are not likely to be queried for, we decided to cut off the corpus terms and the query terms to a length of 12 characters. This resulted in 2,380,958 corpus terms with 12 or less letters, which is a 1.7% decrease in terms.

All letters have been converted to lowercase, which gives an alphabet of 26 letters. The amount of possible letter combinations with terms up to 12 letters is $26^1 + 26^2 + \cdots + 26^{12} \approx 9.9 \cdot 10^{16}$. Five bits are required to encode 26 letters. This means that every 12 letter term uses $5 \cdot 12 = 60$ bits or almost 8 bytes. Eight bytes can encode $2^{64} \approx 1.8 \cdot 10^{19}$ letter combinations, which is much more than is required for the 2,380,958 unique terms. Most letter combinations will never occur in the corpus, because they are not normal terms. Therefore, term identifiers can be smaller than 8 bytes when a mapping is found that only assigns identifiers to terms that are likely to occur in a corpus.

5.1.2 Technical explanation

This section explains how every unique corpus term can have a unique 3 byte identifier, which is useful because it takes up less space than the term. This helps in reducing the size of the local and global indices that store the keys. This section distinguishes between terms with a length of five or less letters and terms with six of more letters.

Terms with five or less letters

An identifier is calculated immediately for every term that has five or less letters. These terms can be mapped to 27^5 numbers, because there are 26 lowercase letters and there is one 'no-token' letter. This 'no-token' letter is used as suffix padding to make every term 5 tokens long. For example, the three letter term `dog__` has three normal letters and two no-token letters to make it five letters in total. The letter `d` is the fourth letter in the alphabet, so it is represented by 4. This can be done for every letter. The 'no-token' is represented by 0. Every term identifier for a five letter term (ti_5) is calculated using the following formula:

$$\text{ti}_5 = (\text{letter1} \cdot 27^4) + (\text{letter2} \cdot 27^3) + (\text{letter3} \cdot 27^2) + (\text{letter4} \cdot 27) + \text{letter5}$$

For the example of the term `dog`, this becomes:

$$\begin{aligned}\text{ti}_5 &= (\texttt{d} \cdot 27^4) + (\texttt{o} \cdot 27^3) + (\texttt{g} \cdot 27^2) \\ &= (4 \cdot 531,441) + (15 \cdot 19,683) + (7 \cdot 729) = 2,426,112\end{aligned}$$

A term identifier that has been calculated using this formula will be called a *five letter encoding*.

Terms with six to twelve letters

Every unique term that consists of six or more letters has a unique term identifier that is assigned in order of term appearance. The first unique term gets the number $27^5 = 14,348,907$. The term identifier is incremented by 1 for every new unseen unique term. An index stores the term identifier assignments and can be used to retrieve the term identifier that represents a term. Three bytes gives 2^{24} possible term identifiers. 14,348,907 of those possibilities are reserved for terms up to five letters, so three bytes contain space for $2,428,309$ term identifiers. This is enough to store every unique term in the WT10g corpus.

Before explaining the index structure, two definitions are introduced. Two consecutive letters of a term is a *duplet*. Three consecutive letters of a term is a *triplet*. The procedure that is used to calculate a five letter encoding can also be applied to calculate encodings for duplets and triplets. For example, the following formula is used to calculate a triplet encoding:

$$\text{ti}_3 = (\text{letter1} \cdot 27^2) + (\text{letter2} \cdot 27) + \text{letter3}$$

The unique term index consists of an array and many small trees that are built using duplets, triplets and five letter encodings. Fig. 5.1 shows a small piece of a unique term index. It shows how the terms `photos`, `phonetic`, `photocell`, `photoflash`, `photography` and `photographer` have been added to the unique term index in such way that the term identifiers can be retrieved for the terms. Terms and their identifiers are put in the unique term index using a procedure that is described below. The description uses examples of calculations that can be found in the figure.

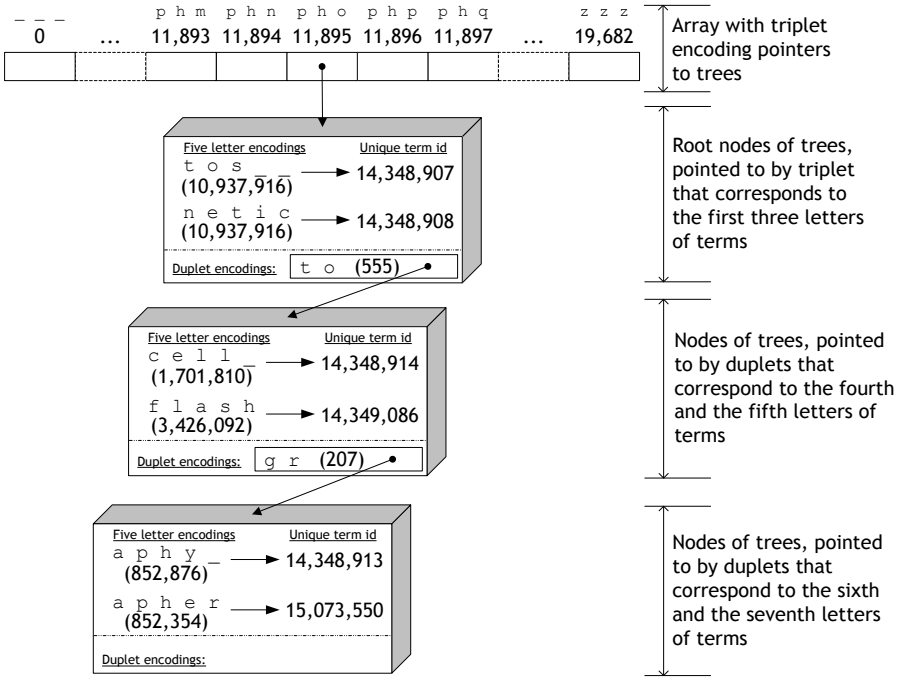

Fig. 5.1. Unique term index showing one branch of one tree that stores term identifiers for the terms `photos`, `phonetic`, `photocell`, `photoflash`, `photography` and `photographer`.

- Calculate the triplet encoding for the first triplet of the term. For example, the first triplet of `phonetic` is `pho` and its encoding is 11,895. The encoding number is used to find a pointer in the array that references to a root node of a tree (as shown in the top of Fig. 5.1). A root tree node is created if it did not exist before and a pointer to it is stored in the array at the position of the calculated triplet encoding.
  Is the term length 6, 7 or 8 characters?
  yes Calculate the five letter encoding for the remaining letters, e.g. the five letter encoding for the remaining letters of `phonetic` (`netic`) is

7,553,415. The five letter encoding is stored in the root node together with a unique term identifier which has the value of the previous highest term identifier value incremented by 1.

no Calculate the duplet encoding for the next set of letters (fourth and fifth letter), e.g. for `photography` the next duplet is `to` and its encoding is 555. A pointer from the root node to a new node and the new node itself are created for the duplet `to` if they did not exist yet.

Is the term length 9 or 10 characters?

yes Calculate the five letter encoding for the remaining letters. Store the five letter encoding in the duplet node, together with a unique term identifier by using the value of the previous highest term identifier and incrementing this value with 1.

no Calculate the duplet encoding for the next set of letters (seventh and eighth letter), e.g. for `photography` the next duplet is `gr` and its encoding is 207. A pointer from the previous duplet node to a new node and the new node itself are created for the duplet `gr` if they did not exist yet.

The final step consists of calculating the five letter encoding for the rest of the term, e.g. the five letter encoding of `photography` is 852,876 (for `aphy_`). This five letter encoding is stored in the duplet node, together with a unique term identifier by using the incremented value of the previous highest term identifier.

Discussion

The procedure description above explained that the longest terms are split in a triplet (letter 1–3), a duplet (letter 4–5) and another duplet (letter 6–7), before a five letter encoding is calculated. There is nothing that prevents this system from using more duplets before calculating a five letter encoding. Therefore, the system is also capable of dealing with longer terms.

The strength of the unique term index lies in exploiting the overlap of term letters to reduce the amount of space required for the term and the term identifier. The index sample showed six terms starting with `pho`. Those three letters need to be stored only once (using an entry in the array). This overlap occurs with many terms and on many levels in the term (first triplet, first duplet and second duplet), which reduces the number of nodes required to store the term identifiers.

We could have chosen to reserve more space in the array for longer letter combinations, e.g. using quadlets. This is a trade-off between speed and storage. There are three factors that come into play with this consideration. Longer first letter combinations results in:

A larger array The array needs space for more pointers. The fill percentage of the array decreases when the number of letters is increased, because many longer letter combinations are not valid starts of terms. The terms of the

WT10g corpus start with 18,177 different three letter combinations (out of $26 \cdot 27 \cdot 27 = 18,954$ possible combinations). This is an array fill percentage of 92%. With 240,946 four letter combinations – out of $26 \cdot 27 \cdot 27 \cdot 27 = 511,758$ combinations – the fill percentage drops to 45%.

Shallower trees and less nodes in the trees The root nodes can store term identifiers for longer terms. This advantage is passed on to underlying nodes, resulting in less deeper nodes for the longest terms.

More trees When the array stores pointers for the first four letters of a term, the root nodes store the term identifiers for all terms with a length between 6 and 9 letters (79% of all unique terms longer than 5 letters in the WT10g corpus). When triplet encodings are used to address the array, all terms with a length between 6 and 8 letters are stored in the root nodes (65% of all unique terms longer than 5 letters in the WT10g corpus).

The space wasted by empty entries in the array is marginal compared to the space requirement for extra nodes; a node needs to store tuples of five letter encodings and term identifiers and it needs to store tuples of duplet encodings with pointers to child nodes. The node structure itself uses more space than an unused array entry (which has the size of one pointer). Therefore, the trick is to keep the amount of nodes low, while also ensuring that the average number of encodings in a node stays low (otherwise the retrieval of term identifiers will take too much time).

The optimal settings cannot be calculated, because the settings are dependent on the diversity of the terms in the corpus. The optimal settings can be obtained by experimentation. Our experiments showed that the use of triplets for array addressing resulted in enough trees. Together with the use of duplets, the overlap of terms was exploited such that the memory footprint was acceptable with a 50MB unique term index. For reference, the size of the plain text file with all unique terms is 20MB. The retrieval rate was 4.5 million term identifiers per second on a 2.5GHz machine.

5.1.3 Alternatives

Hashing

An alternative for the unique term index is a hash function that maps terms to term identifiers. A hash function maps every term to a 32-bit integer. This integer can be used to find a data item (term identifier) that is associated to the term. This integer based search can be performed much faster than a term based search.

A 12 letter term with lower case letters requires 60 bits for encoding. Mapping these 60-bit encodings to 32 bits can cause duplicates; two different terms can be hashed to the same 32-bit integer. The entire term needs to be stored to detect these “hash collisions”. Hash collision tests have to be performed for every calculated hash. If a collision occurred, it needs to be resolved by finding

the entry that corresponds to the original term. This typically requires one extra lookup. Hash collision do occur with 2,380,958 terms, even though there are 4.3 billion integers to map terms to. The C# hash function has 691 collisions with 2,380,958 unique terms, so storing the actual terms is really required.

The average corpus term length is six letters and can be encoded with ASCII. This means that the average term uses 10 bytes: 6 bytes for the letters and 4 bytes for an integer denoting the string length. Every term has a unique term identifier which is stored as an integer (4 bytes). This means that every term uses 14 bytes. With 2,380,958 unique terms, the data itself uses 33MB. Additional storage is required for the hash table structure that stores the terms and their identifiers. The total amount of data comes down to 128MB for 20MB of unique terms. The storage requirement is more than twice as much as the unique term index. Term identifier lookups are slower than with the unique term index: 3.9 million term identifiers per second on a 2.5GHz machine. We used a C# test implementation to obtain these numbers.

Perfect hashing

A perfect hash function is a hash function that has no collisions when mapping terms to integers. It takes about one hour on a 1 GHz machine to create a perfect hash function for 2,380,958 keys using gperf [62]. The perfect hash function calculation can be sped up by distributing it over many machines; Botelho *et al.* can calculate a perfect hash function for 1.024 billion keys in four minutes using 14 commodity PC's [8].

A new perfect hash function will have to be calculated when new unique terms are added. This means that all indices will have to be updated as well. This requires that all unique terms in the index have to be stored separately, because hash functions are one way functions. Using the list of unique terms, the old hashes can be calculated and converted to new hashes using the new hash function.

Perfect hash functions are larger than regular hash functions, but the total memory footprint can be smaller. Perfect hash functions can be an interesting option if the memory usage should be minimized.

5.2 Multi-level term index

The multi-level term index is an index structure that stores term sets with additional information about those terms sets. The type of additional information depends on the application of the multi-level term index. The index can be used to store information about a local collection or it can be applied to store global information about collections at a broker. This section starts with a description about the technical structure of a multi-level term index and is followed by the two applications for the index structure. The section concludes with a discussion about alternatives for this index.

5.2.1 Structure of the index

The structure of the multi-level term index is comparable to the unique term index, as it also uses an array with relatively shallow trees to store nodes. The difference is that the multi-level term index uses the nodes to store information about the terms, instead of information about letters of a term. One branch in a tree corresponds to one set of terms.

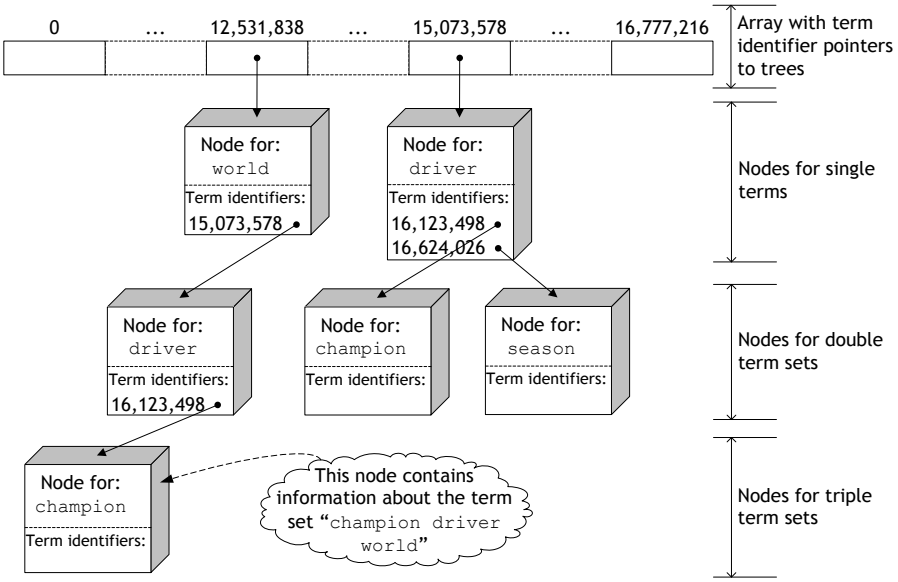

Fig. 5.2. Small part of a multi-level term index that shows how term sets are stored.

A small part of a unique term index is depicted in Fig. 5.2. The figure shows that the array can store 16,777,216 entries. This is equal to the amount of numbers that can be encoded with 3 bytes (2^{24}), and it is the maximum value of a term identifier. Storing and retrieving term sets and associated information in the multi-level term index does not differ much from storing and retrieving term identifiers in the unique term index. The concept is explained using an example of inserting information about the term set "`driver world champion`".

Using the unique term index, term identifiers can be obtained for each of the three terms. The term identifier for `world` can be calculated, as the term length is 5 characters: the term identifier is 12,531,838. The term identifiers for `driver` and `champion` were assigned in order of addition to the unique term index. The same will hold for any other term that has a length longer than five letters. The following terms and term identifiers are used in Fig. 5.2: `champion` (16,123,498), `driver` (15,073,578) and `season` (16,624,026).

The term identifiers are sorted once they have been retrieved from the unique term index. This means that the order of terms in a term set does not matter, i.e.

for Sophos "`driver world champion`" and "`champion driver world`" are the same. This reduces the index size and it speeds up search, because one lookup suffices. However, it can negatively affect collection selection performance when one particular ordering appears frequently in many collections (in at least as many collections as defined by the collection maximum parameter) and another ordering appears less frequent in the collections. The less frequent ordering can not be found in the term order independent approach.

The ordered term identifiers (12,531,838; 15,073,578; 16,123,498) for the term set "`driver world champion`" can be used to traverse the tree and, if non-existent, construct the nodes and pointers along the way. This is done in the same way as with the triplet and duplet encodings from the unique term index. Every node stores pointers to child nodes. Every node can store additional information that applies to that particular term set. For example, Fig. 5.2 shows a node for `driver` with one child node. That node contains information for the term set "`world driver`"; it contains no information about longer term sets (other than the pointers to the nodes that can make longer term sets).

5.2.2 Contents of the index nodes

The nodes in the trees of the local indices contain one term set frequency counter. It counts the occurrences of the term set that is represented by that node (for example, a node on the third level in the tree represents a triple term set). The term set frequency counters need to be able to store large numbers, because single terms can occur very often (even though stop terms have been removed from the corpus); a counter is a four byte integer.

The nodes in the global index store collection identifier counters. As the global index only contains infrequent term sets (with occurrences below the term set frequency maximum), the counters can be smaller: two bytes can already count up to 65,536 occurrences. This is enough for our tests, as the largest tested term set frequency maximum is 750 occurrences. Section 3.1 introduced the collection maximum. This defines the maximum number of collection identifier counters in a global index node.

5.2.3 Alternatives

Storing and quickly accessing large amounts of information is a common problem in the area of Information Retrieval. A popular solution is the use of a database. This section starts by exploring the option of a relational database as an alternative to the multi-level term index and concludes with a discussion about the search cost with tree structures.

Database

Table 5.1 shows a sample of a database that stores global index information for single, double and triple term sets. The byte size of the attributes is shown between brackets. The database size can be reduced in two ways:

- When the term set consists of one term, the tuple will still use the space for three terms. Separate tables for each amount of terms in the term sets will solve this problem.
- There is a lot of redundancy for the term set identifiers. This can be solved by replacing the collection identifier and frequency counter attributes by one attribute, which contains a variable amount of collection identifier counters. A variable size attribute needs some bytes to store the size of a data item, but this is a small overhead compared to the largely reduced amount of tuples in the table. We will assume two bytes to indicate the number of collection identifier counters in the data item.

term_id_1 (3)	**term_id_2** (3)	**term_id_3** (3)	**collection_id** (2)	**frequency_counter** (2)
12,531,838			5	742
12,531,838			6	692
12,531,838	15,073,578		1	742
12,531,838	15,073,578		2	321
12,531,838	15,073,578	16,123,498	3	173
12,531,838	15,073,578	16,123,498	4	432

Table 5.1. Example of database content for global index

Table 5.2 up to Table 5.4 show the improved database tables using the suggestions from above. Instead of $6 \cdot 13 = 78$ bytes, the data would require $13 + 16 + 19 = 48$ bytes.

term_id_1 (3)	**collection_id_counters** (variable size)
12,531,838	2:3-173;4-432

Table 5.2. Example of database content for global index that store single term sets. The data uses $3 + 2 + 8 = 13$ bytes.

term_id_1 (3)	**term_id_2** (3)	**collection_id_counters** (variable size)
12,531,838	15,073,578	2:3-173;4-432

Table 5.3. Example of database content for global index that store double term sets. The data uses $3 + 3 + 2 + 8 = 16$ bytes.

Simply putting all information in tables is not enough, as a good index is required to quickly retrieve the stored information. This index will take up a lot of additional space and will have to be kept up to date. Furthermore, tables are unable to exploit overlap in term sets to reduce the amount of stored data. The table shown in Table 5.1 stores the term identifier 12,531,838 six times. The

term_id_1 (3)	**term_id_2** (3)	**term_id_3** (3)	**collection_id_counters** (variable size)
12,531,838	15,073,578	16,123,498	2:3-173;4-432

Table 5.4. Example of database content for global index that store triple term sets. The data uses $3+3+3+2+8=19$ bytes.

improved database tables stores the same identifier three times. The multi-level term index will store that identifier only once. Note that this was a small sample of a database; a real global index can easily contain tens of millions of term sets, so the space reduction achieved by exploiting overlap is significant.

Tree structures

Tree structures do not need an additional index and can maximize the overlap of term identifiers. This will result in a space efficient solution to store term sets with corresponding information. There are many tree types, such as B+ trees, binary trees, red-black trees and radix trees.

The multi-level term index, depicted in Fig. 5.2, can also be looked upon as a tree. The array can be considered as the "root node" that contains no additional information. All nodes have a theoretical maximum of 16,777,216 branches, but for our tests the maximum number of branches (in the root node) is 2,380,958, as that is the amount of unique terms in the corpus. In practice, the non-root nodes of frequently occurring terms will contain several hundreds of child nodes, because the number of term combinations is limited when the window size is small. This means that those nodes can have a list with hundreds of term identifiers. Searching term identifiers in these lists is the largest search cost component with the multi-level term index.

The search cost for term set associated information can not be calculated when the term set contains more than one term, because the number of child nodes is highly dependent on the text and the window size. We assume the following amount of term sets in a global index: 500,000 single terms; 5,000,000 double terms and 5,000,000 triple terms. Some single terms are infrequent by themselves, so they will have no child nodes. Other single terms are frequent and will have many children to make infrequent multi-term sets. This also applies to term nodes on the second level in the multi-level term index. The lack of precise numbers means that we can only make a worst case estimate for a triple term search.

The first term of a term set can always be found instantly by one lookup in the array. The search cost for the next terms depend on the number of term identifiers in the node. We assume that a popular term has a set of 500 term identifiers with pointers to child nodes. Binary search is used to find the child node pointer corresponding to the second term in the term set. Approximately $\log_2(500) \approx 10$ comparisons are needed to find the pointer that leads to the next child node. The same amount of operations is required for every next term in the term set. The multi-level term index can have an unlimited number of children,

which means that the structure can store term sets with an unlimited number of terms. This is where the multi-level term index differs from other tree structures.

Most regular tree structures are well suited for fast lookup of term associated information when the number of unique terms is very large, but there are only single terms as keys. For example, a B+ tree with a fan-out of 8 can store 16,777,216 term identifiers in 8 levels. The term related information is stored in the leaf nodes at the bottom level. The worst case search requires 24 comparisons to find the term associated information, because $\log_2(8) = 3$ comparisons are required per level. The average case search requires $8 \cdot 2.5 = 20$ comparisons. This is worse than the amount of comparisons required with the multi-level term index.

A key can also contain multiple terms, which makes it harder to apply traditional tree structures. There are two options to store multiple terms with trees:

1. Increase the key space. Every term can be encoded with 24 bits. A term set can consist of three terms, resulting in a key space with $2^{3 \cdot 24}$ possibilities. The term information can be indexed using B+ tree with a fan-out of 16 and 18 levels. The worst case search needs 72 comparisons, because $\log_2(16) = 4$ comparisons are required per level. The average case search requires 3.3 comparisons per level, so 60 comparisons are required on average. This number of comparisons is required for single, double and triple terms. This solution does not scale well when the maximum number of terms in a term set is increased.
2. Use multiple levels of trees, similar to the multi-level term index. A first tree is used to find the node that corresponds to the first term. This node can point to a tree that stores the nodes for the second terms. This concept can be extended to store an unlimited number of terms in a term set. The search costs are multiples of the previously calculated search costs, e.g. a term set with three terms needs 72 comparisons in worst case and 60 comparisons in the average case. This is the same as the previous solution, but the advantage is that single term searches still take 24 comparisons in worst case.

We have shown that the multi-level term index is faster for single term searches than a B+ tree: a worst case of 1 array lookup versus 24 comparisons. Multi-term searches are also performed with less comparisons, with an estimated 10 comparisons per term for the multi-level term index versus 24 comparisons for the B+ tree.

5.3 Conclusion

This chapter described the Sophos index structure. It introduced the unique term index as a data structure to create small 3-byte representations for the terms. The unique term index appeared to be a better alternative than hashing in terms of retrieval speed (4.5 million versus 3.9 million lookups per second) and storage requirements (50MB versus 128MB).

The same conclusions were drawn with respect to the multi-level term index for storing term sets and their associated information. The multi-level term index was compared to relational databases and the B+ tree. The multi-level term index is better capable of exploiting the term overlap in term sets than a relational database. Furthermore, the storage cost for a relational database will exceed that of the multi-term index, because the relational database stores the index and data separate, which introduces additional overhead. Compared to the B+ tree, the multi-level term index is capable of retrieving term sets in less operations.

6

Results analysis

Don't tell people how to do things. Tell them what to do and let them surprise you with their results.

– George Patton (1885–1945)

This chapter describes the results of all experiments. In total, 8400 separate experiments have been executed. For readability purposes, only a selection of the results of those experiments is presented. This chapter is divided in different topics. Each topic starts with a selection of relevant results – if they are required and not already presented – and is followed by a discussion. The first topic is about corpus splits and presents the best results that were obtained by the different retrieval methods. That section will often be referred to while discussing the other topics.

6.1 Points of attention before looking at the graphs

Abbreviations are used in the graphs to keep them surveyable. The abbreviations are listed in Table 6.1. An example that can be found in several graphs is tf250cm5. This means that the parameter setting of 250 has been chosen as term set frequency maximum, combined with a collection maximum of 5.

cm	Collection maximum
ColID counters	Collection identifier counters
ColRank	The PageRank-like collection ranking algorithm
LMDS	Language modeling with Dirichlet smoothing
QDI	Query-driven indexing
QDI AOL	Query-driven indexing using the AOL query log
QDI Combined	Query-driven indexing using the AOL query log combined with ColRank
tf	Term set frequency maximum
ws	Window size

Table 6.1. Abbreviations used in result graphs

It is important to note that all precision related graphs in the following section have a range of 0 to 0.5, except for the IP Split and IP Merge 100 graphs that involve navigational queries.

Sometimes it seems that the result graphs for ColRank or QDI Combined appear to be missing. In this case, those two graphs are exactly the same and they overlap each other.

6.2 Corpus splits

This section discusses the effects of the two types of corpus splits on the collection selection performance. First the results with three random corpus splits are presented. Those results are concluded by a discussion on the collection selection performance. Second, the IP based corpus split results are presented and followed by a discussion about them.

Per corpus split, four graphs are shown. The first two graphs cover the navigational queries, the second two graphs cover the informational queries. Every pair consists of a precision graph and a cumulative recall graph. The precision was calculated using Formula 4.3. The recall was calculated using Formula 4.1. All graphs are obtained by taking averages for multiple queries: 145 queries for the navigational queries, 50 for the informational queries. Every collection selection system produces a ranked result list per query. Every graph shows the average precision or recall at position n of those result lists, where n is the number of selected collections up to that point.

6.2.1 Random splits

Random 100 result graphs

Fig. 6.1 shows the precision for the navigational queries. The graph shows that every collection selection method is unable to achieve a high precision. The average precision is 0.025 when all collections are selected ($n = 100$). This means that there is an average of 2.5 relevant collections in the Random 100 corpus split for the 145 navigational queries. All precision figures are close to this value. ColRank and the combination of query-driven indexing with ColRank have the exact same precision figures.

Fig. 6.2 shows the recall for the navigational queries. ColRank and the combination of query-driven indexing with ColRank seem to approximate the linear line $y = x/100$. The two have the same recall figures. LMDS and HDKs with query-driven indexing are above that line, LMDS being the best.

Fig. 6.3 shows the precision for the informational queries. The average precision is 0.19 at $n = 100$, i.e. on average there are 19 relevant collections for the 50 informational queries. ColRank, and the combination of query-driven indexing, have a precision that stays lower than 0.19. The two are exactly the same.

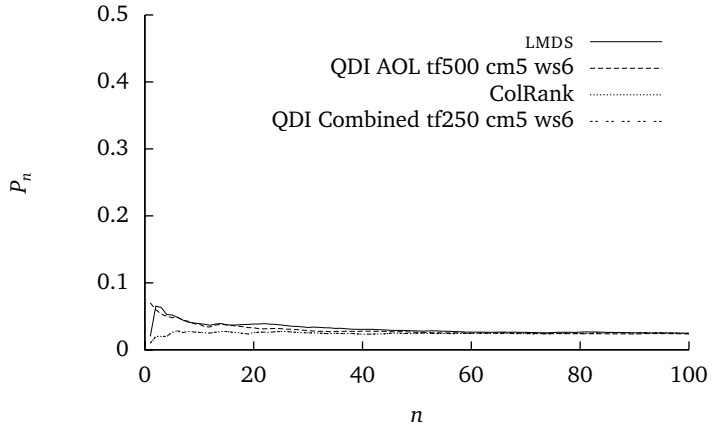

Fig. 6.1. Corpus split: *Random 100*; query type: *navigational*; graph: *precision*.

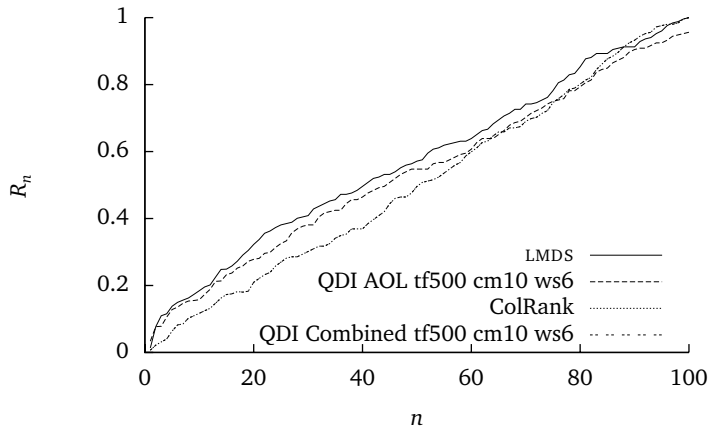

Fig. 6.2. Corpus split: *Random 100*; query type: *navigational*; graph: *recall*.

LMDS and query-driven indexing are better, but looking at the average number of relevant collections, the precision is not that high as well.

The recall for the informational queries is depicted in Fig. 6.4. Both ColRank and the combination with query-driven indexing seem to behave linear from $n = 13$. The two are exactly the same. Query-driven indexing has a better recall for the first 14 selected collections. LMDS is better from $n = 14$ with a slightly higher recall. All graphs appear to increase linear from $n = 20$.

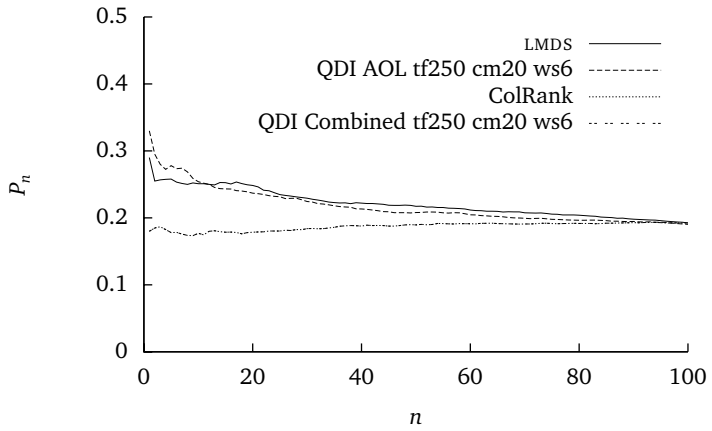

Fig. 6.3. Corpus split: *Random 100*; query type: *informational*; graph: *precision*.

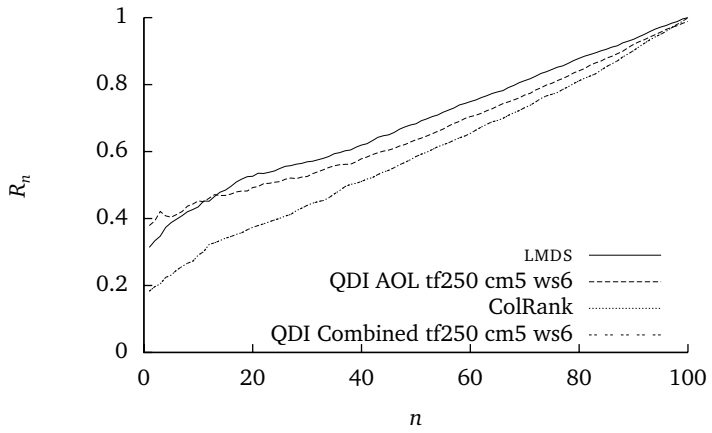

Fig. 6.4. Corpus split: *Random 100*; query type: *informational*; graph: *recall*.

Random 11512 result graphs

Fig. 6.5 shows the precision for the navigational queries. The precision achieved by LMDS starts at 0.05 and drops from there on. This score may give the impression that precision is dramatically bad, but this conclusion is too premature. There are 11,512 collections and approximately 2 relevant collections per query. Random collection selection would have a constant precision value around 0.0002. This is explained in more detail in Appendix C.

Fig. 6.6 shows the recall that is achieved for the same test setting. This graph shows that LMDS and QDI AOL are in fact able to select relevant collections, as the recall keeps increasing. LMDS has selected half of the available merit after

selecting 1000 collections. Such numbers indicate that all collection selection methods are not very useful, but they also show that the precision graph from Fig. 6.5 may be misleading at first glance.

The two graphs also show the difference between no query-driven indexing and query-driven indexing. Pruning keys from the index decreases selection performance with this experiment.

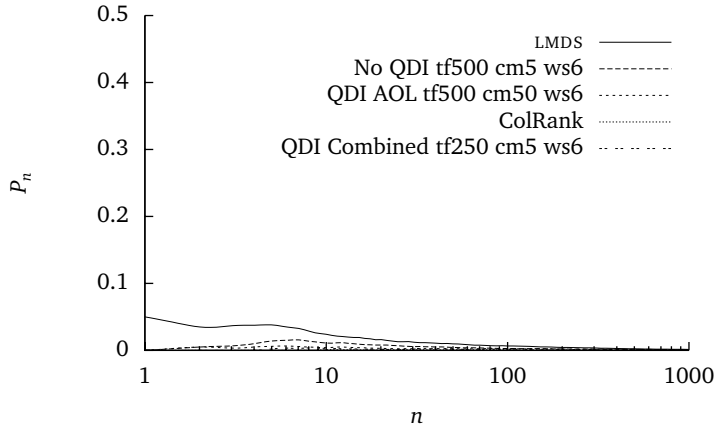

Fig. 6.5. Corpus split: *Random 11512*; query type: *navigational*; graph: *precision*.

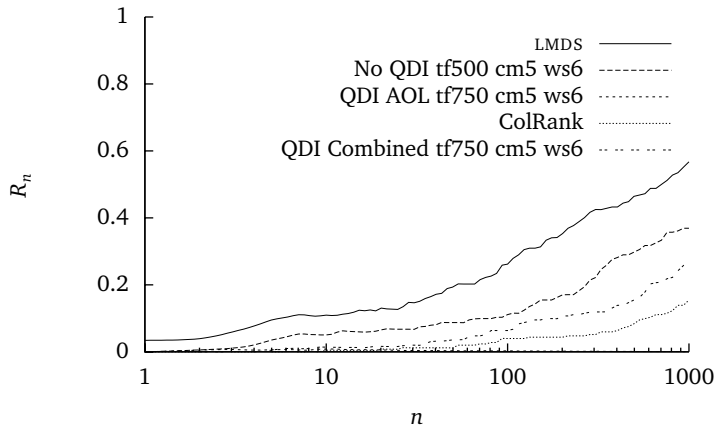

Fig. 6.6. Corpus split: *Random 11512*; query type: *navigational*; graph: *recall*.

The precision for the informational queries is depicted in Fig. 6.7. The graph shows that LMDS is at least twice as precise as all other collection selection sys-

tems. This is also reflected in the recall graph that is shown in Fig. 6.8. The two graphs also shows the effect of pruning keys from the index (No QDI versus QDI AOL). Pruning decreases selection performance within the first ten selected collections, but the difference is smaller than with the navigational queries. Once 90 or more collections were selected, selection performance also decreased.

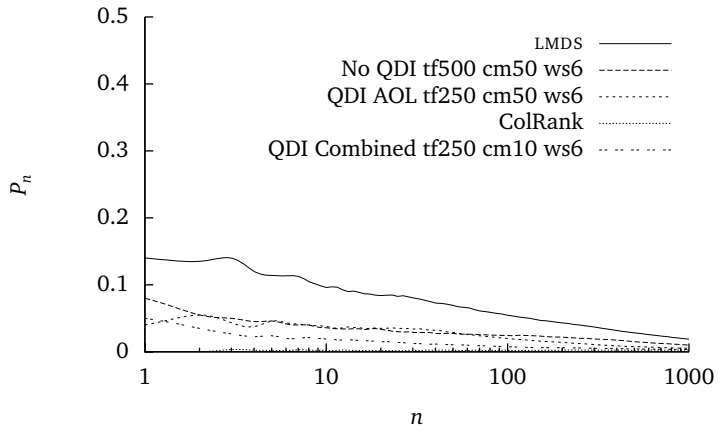

Fig. 6.7. Corpus split: *Random 11512*; query type: *informational*; graph: *precision*.

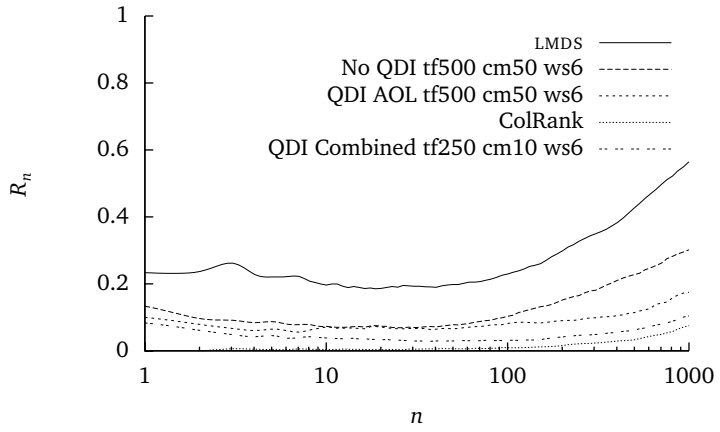

Fig. 6.8. Corpus split: *Random 11512*; query type: *informational*; graph: *recall*.

Overlap3 11512 result graphs

Fig. 6.9 shows the precision for navigational queries. The graph shows that LMDS performed best. Using this large corpus split, it was impossible to run the test with normal query-driven indexing. Therefore, the test was run with query-driven indexing that pruned all keys that occurred less than two times in the query log. The selection performance is less good than with LMDS, but much better than that of ColRank. ColRank has a precision close to 0. The same relation between the three collection selection systems is also reflected in the recall graph for navigational queries, shown in Fig. 6.10.

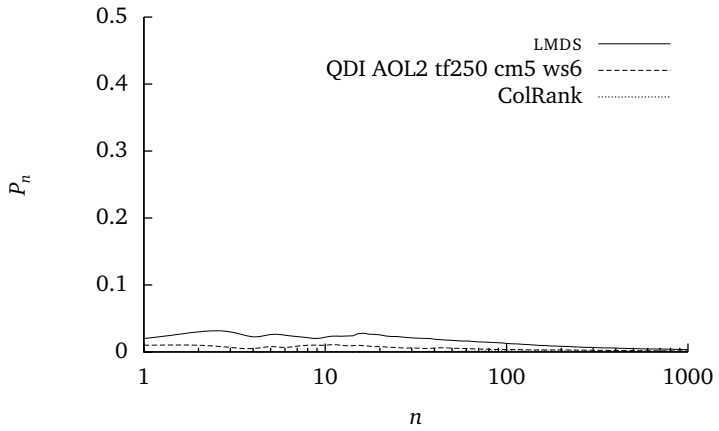

Fig. 6.9. Corpus split: *Overlap3 11512*; query type: *navigational*; graph: *precision*.

Fig. 6.11 and Fig. 6.12 respectively show the precision and recall for the informational queries. When the two graphs are compared with the previous two graphs for navigational queries, the relations between the three collections selection systems remain unchanged: LMDS is the best, followed by QDI AOL2 and tailed by ColRank. But the difference is that precision and recall start much higher for informational queries than for navigational queries.

Discussion on random corpus split results

When documents are randomly assigned to the collections, we expect an equal distribution of the merit in the sense that there are no high merit collections. Some collections have most query terms and are relevant, while other collections will contain some query terms but are not relevant. These distribution properties require that a collection selection system must have good knowledge of the exact contents of each collection to be able to achieve high precision. The cumulative recall will also be high when the precision is high, because random corpus splits have low merit collections.

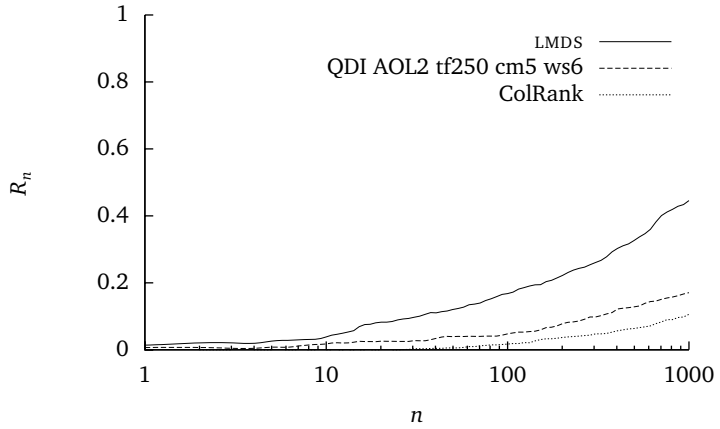

Fig. 6.10. Corpus split: *Overlap3 11512*; query type: *navigational*; graph: *recall*.

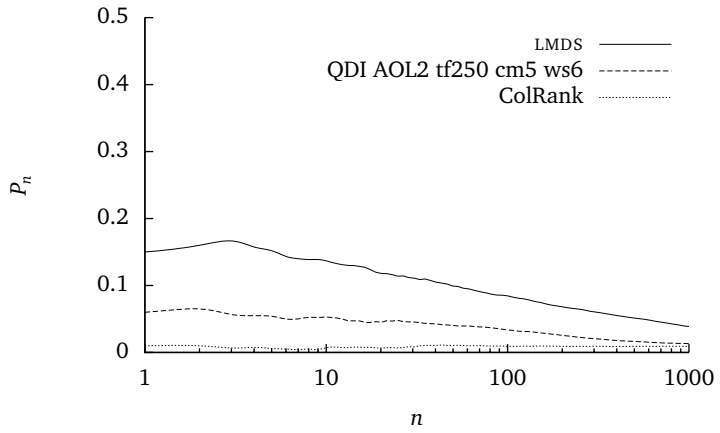

Fig. 6.11. Corpus split: *Overlap3 11512*; query type: *informational*; graph: *precision*.

Fig. 6.5 up to Fig. 6.12 show that LMDS performs better with random splits when the number of collections is large (11,512 collections). LMDS is a centralized system that has all knowledge about all collections, so this result could be expected; it knows exactly which collection contains which query terms.

LMDS is not always better than query-driven indexing when the number of collections is smaller (100 collections). When LMDS is better with the Random 100 corpus split, it is not much better than query-driven indexing, as is illustrated by Fig. 6.1 up to Fig. 6.3. This supports the conclusion that collection selection on a random corpus split with a small amount of collections is very hard. The collections probably contains too many different topics, and too many similar topics are spread over different collections.

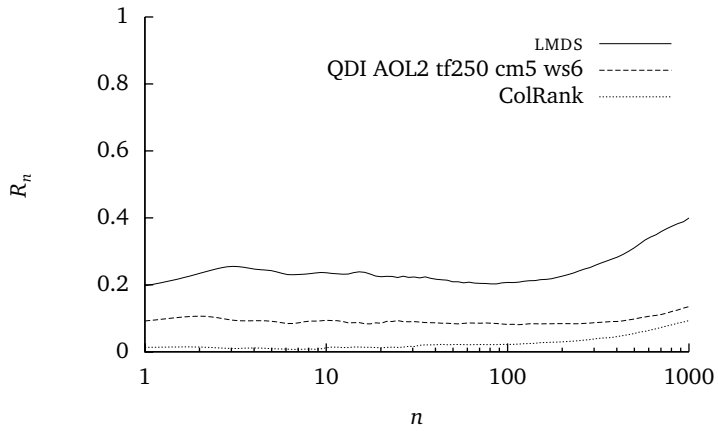

Fig. 6.12. Corpus split: *Overlap3 11512*; query type: *informational*; graph: *recall*.

Not many tests have been done with the Overlap3 11512 split, because the split was simply too large for most systems. When comparing the Overlap3 11512 with Random 11512, we expect the following effects on retrieval performance:

- Precision can be higher for Overlap3 11512, because there are more relevant collections.
- Precision can be lower for Overlap3 11512, because more query terms can co-occur at non-relevant collections.
- As there are more collections with merit, and merit is still distributed equally, the cumulative recall will show better results.

The Overlap3 11512 collection selection figures were generally worse than with Random 11512. This was unexpected for the recall graphs. The cause may be found in the random assignment of documents. The experiment should be repeated to verify whether this effect is representative for all random corpus splits.

6.2.2 IP based splits

IP Merge 100 result graphs

Fig. 6.13 and Fig. 6.14 respectively show the precision and recall for the navigational queries with IP Merge 100. The precision graph shows that LMDS is very capable of selecting the most relevant collection. This is especially good as the average number of relevant collections per query is two. The rapid decrease in precision is not misbehavior of LMDS; there are simply no more relevant collections to select with such a good start.

It is interesting to compare the two graphs with their Random 100 counterparts (Fig. 6.1 and Fig. 6.2). The relations between the collection selection

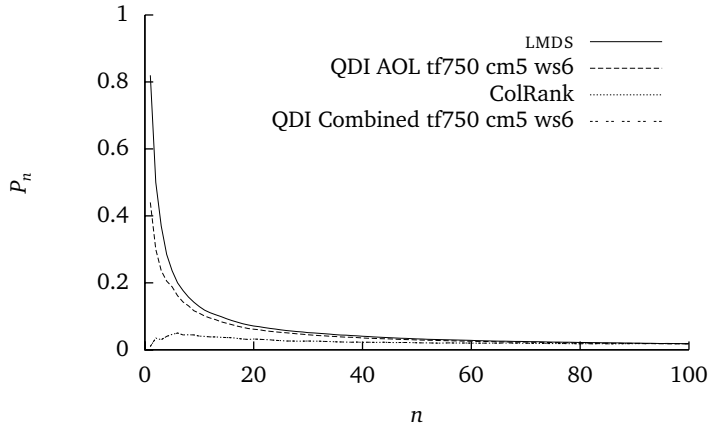

Fig. 6.13. Corpus split: *IP Merge 100*; query type: *navigational*; graph: *precision*.

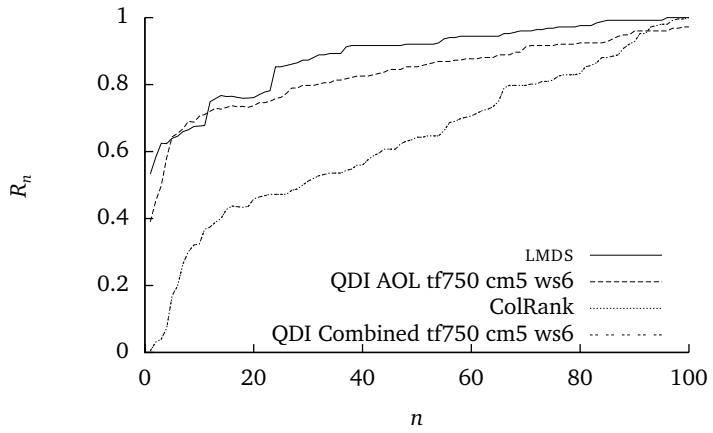

Fig. 6.14. Corpus split: *IP Merge 100*; query type: *navigational*; graph: *recall*.

systems are the same, but both precision and recall are much higher with the IP Merge 100 corpus split.

Fig. 6.15 and Fig. 6.16 respectively show the precision and recall for the informational queries with IP Merge 100. The graphs show that ColRank comes much closer to LMDS and query-driven indexing in terms of precision and recall after selecting 10 collections. The difference is big within the top 10 of selected collections. The top 10 is the most important part to perform well as user queries should not be forwarded to too many collections for query answering in distributed information retrieval.

Once again, it is interesting to compare the precision and recall figures to the Random 100 informational counterpart that are shown in Fig. 6.3 and Fig. 6.4.

The absolute differences in precision and recall are not so big between the two corpus splits: recall is higher with the IP Merge 100 corpus split. But a closer look at the precision graphs show that the Random 100 corpus split has approximately 19 relevant collections per query, against 12 for the IP Merge 100 corpus split. Based on the number of relevant collections, collection selection with the IP Merge 100 corpus split should be a harder task, but precision and recall figures are better.

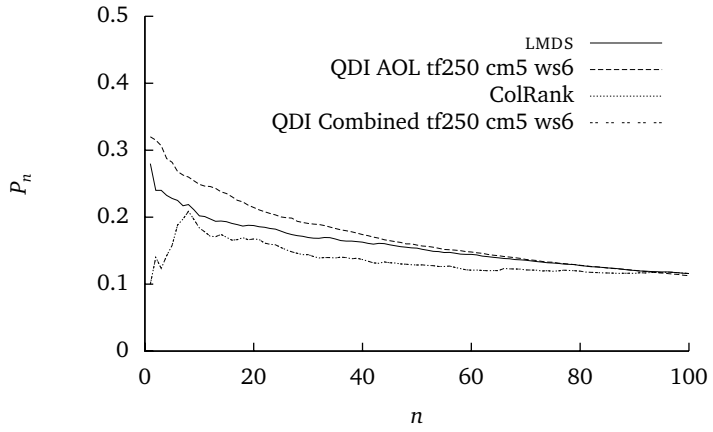

Fig. 6.15. Corpus split: *IP Merge 100*; query type: *informational*; graph: *precision*.

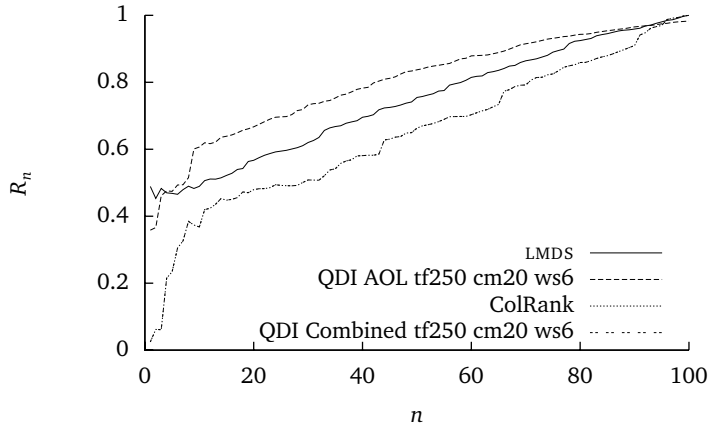

Fig. 6.16. Corpus split: *IP Merge 100*; query type: *informational*; graph: *recall*.

IP Split result graphs

Fig. 6.17 shows the precision for navigational queries. LMDS is the best, having precision values that are at least twice as high as the best query-driven indexing approach. The query-driven indexing approaches all have precision values that are relatively close together. They follow the trend that when the amount of pruned keys increase, the precision drops. ColRank is also plotted, but has a precision that is nearly 0 for every n. The combination of query-driven indexing with ColRank is much better than ColRank alone, but it is not better than the least performing query-driven indexing approach.

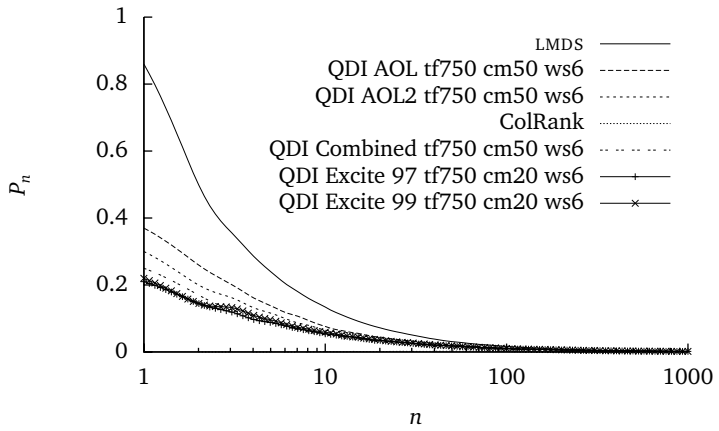

Fig. 6.17. Corpus split: *IP Split*; query type: *navigational*; graph: *precision*.

The recall graph for the navigational queries is shown in Fig. 6.18. The recall figures show the same relations between the collection selection systems as with the precision figures: LMDS is the best, ColRank is the worst, all variations of query-driven indexing are somewhere in between. Within the top 10 selected collections, LMDS has collected 80% of the merit that it could have collected up to that point against 50% for the normal query-driven indexing approach.

Fig. 6.19 shows the precision for the informational queries. The graph shows that query-driven indexing is able to achieve the highest precision. Two things are noteworthy in the graph. The first is that query-driven indexing with the (much smaller) Excite query logs produces results that are almost as good as the other query-driven indexing approaches. The second is the precision of LMDS, which is worse than with navigational queries (shown in Fig. 6.17 with a different y-axis range).

The recall graph for the informational queries (shown in Fig. 6.20) is more suited for showing the mutual relations between the collection selection systems than Fig. 6.19. The mutual relations are comparable for the precision and recall graphs, except for query-driven indexing approaches that use Excite query logs.

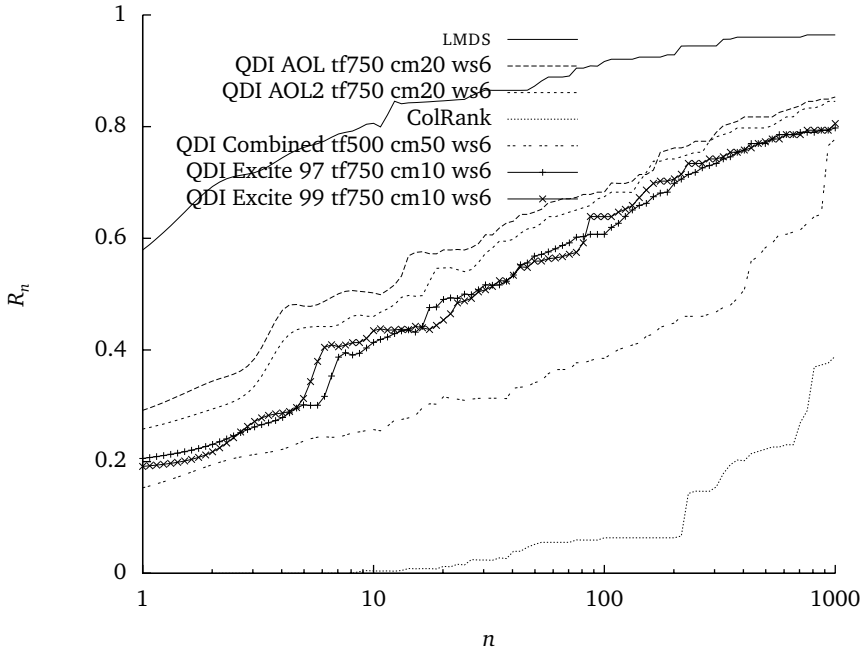

Fig. 6.18. Corpus split: *IP Split*; query type: *navigational*; graph: *recall*.

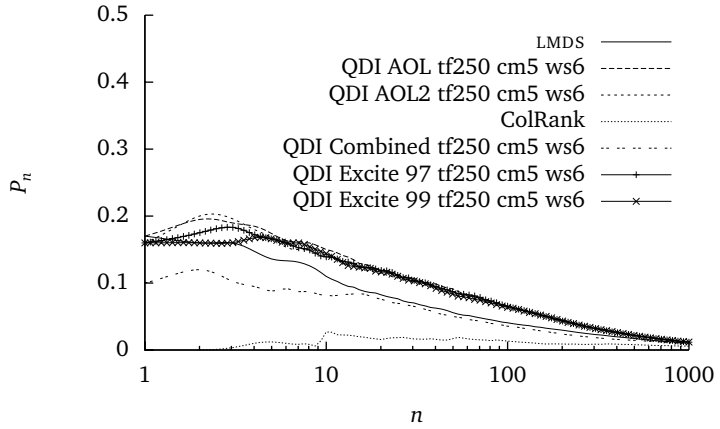

Fig. 6.19. Corpus split: *IP Split*; query type: *informational*; graph: *precision*.

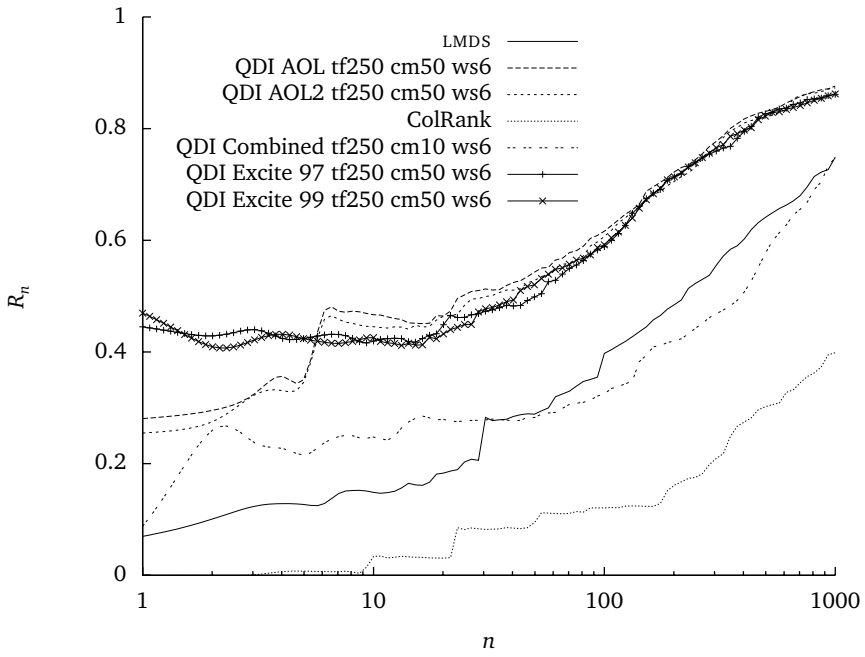

Fig. 6.20. Corpus split: *IP Split*; query type: *informational*; graph: *recall*.

Discussion on IP based corpus split results

A side effect of IP based document clustering is that relevant documents are clustered, as shown in Appendix B: *Merit in corpus splits*. This is especially true for informational queries, where the split resulted in low and high merit collections. It is important for a collection selection system to be able to identify collections that cover a topic that is queried for, i.e. is able to identify collections with many relevant documents (high merit). We expect that this is more important than finding the most relevant document, as the most relevant document can also reside in a collection with low merit. For navigational queries – where there is a little number of relevant collections – we expect that it is more important to be able to find the relevant documents in the collections.

Looking at the collection selection results, HDKs in combination with query-driven indexing is the best system for answering informational queries in IP based splits. LMDS performs exceptionally well with navigational queries. This can be contributed to the fact that LMDS has all the terms in its index which help in selecting the low number of relevant collections – typically less than 5. But this does not explain why the precision and recall degraded with LMDS for the informational queries. Query-driven indexing appears insensitive to the query types as it performed constant on both types. This means that query-driven indexing

did not perform better on informational queries, but LMDS's performance was not as good as expected.

Finally, precision and recall was higher for IP based corpus splits than with random corpus splits.

6.3 Query-driven indexing

Results

No query-driven indexing versus query-driven indexing

Fig. 6.21 shows the number of term sets in the global index for the Random 11512 corpus split. It shows the numbers for no query-driven indexing versus standard query-driven indexing. The parameters with standard query-driven indexing are a term set frequency maximum of 500 and a collection maximum of 20. All graphs in this section use these query-driven indexing settings. The number of term sets are split in single terms, double terms and triple terms. The graph shows that the number of term sets is reduced by almost 95% by applying query driven indexing.

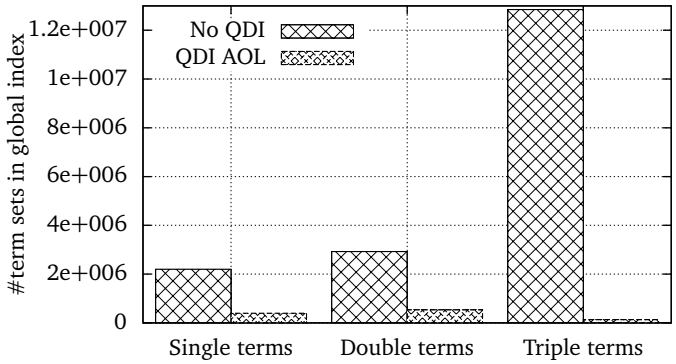

Fig. 6.21. Global index size by number of term sets stored per QDI strategy for Random 11512 corpus split

Instead of the number of term sets, Fig. 6.22 shows the number of collection identifier counters in the global index. The graph shows a 70% reduction in number of collection identifier counters when query-driven indexing is applied. The proportional reduction is less than with the number of term sets.

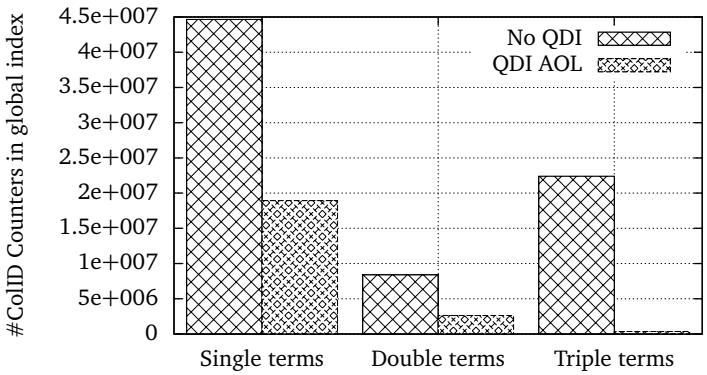

Fig. 6.22. Global index size by number of collection identifier counters stored per QDI strategy for Random 11512 corpus split

Varying query-driven indexing strategies

The index size is reduced by pruning keys in the index that do not occur in a query log. Several pruning strategies have been tested in this research: pruning has been done using the query logs from AOL (QDI AOL), Excite 1997 (QDI Excite 97) and Excite 1999 (QDI Excite 99). Furthermore, the index has been pruning using the AOL query log by removing all keys from the index that occur less than two times in the query log (QDI AOL2). Fig. 6.23 and Fig. 6.24 respectively show the number of terms sets in the global index and the number of collection identifier counters in the global index for those different strategies. The numbers are obtained by analyzing the global indices that were created for experiments on the IP Split. The graph shows that QDI AOL2 results in less term sets and collection identifier counters than using standard query-driven indexing (QDI AOL). The use of the Excite query logs results in a larger reduction. The two graphs show a decrease in number of term sets and number of collection identifier counters that is proportional to the chosen query-driven indexing strategy.

Discussion

Query-driven indexing has been introduced to reduce the index size. This is required, as the number of generated HDKs explodes. A good estimator for the actual index size is the number of collection identifier counters, because all counters together take up the most space in the global index. Fig. 6.22 showed that the reduction was almost 70%, which means that the index size will be reduced by almost the same amount. The index size reduction comes at the expense of collection selection performance. Fig. 6.7 and Fig. 6.8 show that the loss in precision and recall is minimal for informational queries. Fig. 6.6 shows that the loss

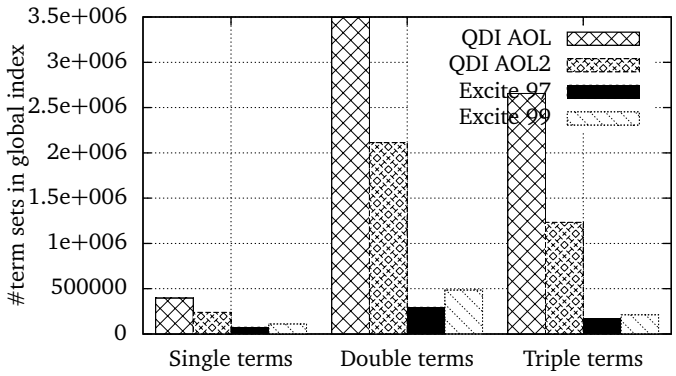

Fig. 6.23. Global index size by number of term sets stored per QDI strategy for IP Split

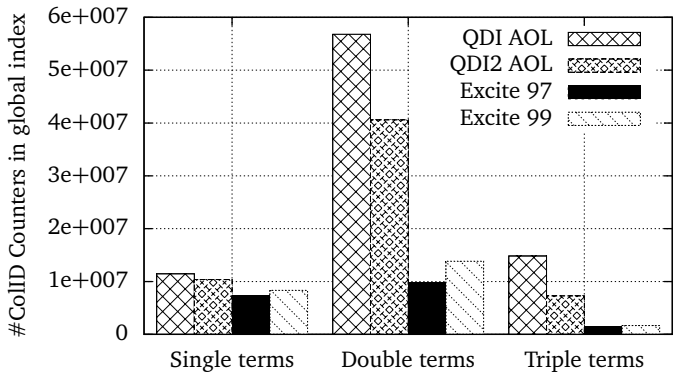

Fig. 6.24. Global index size by number of collection identifier counters stored per QDI strategy for IP Split

in recall for the navigational queries is much; almost no merit has been collected at $n = 10$ by the query-driven indexing approach. At $n = 100$, the query-driven indexing approach has collected about half the amount of merit compared to no query-driven indexing.

The sizes of the indices generated by the different query-driven indexing strategies can be estimated using the number of collection identifier counters that are depicted in Fig. 6.24. Even though not all strategies have been employed on every corpus split, a trend emerged from the results: pruning more keys results in a decrease of collection selection performance. This can be seen by looking at the precision and recall for navigational queries on the IP Split, shown in Fig. 6.17 and Fig. 6.18. It shows that QDI AOL is better than QDI AOL2, and QDI AOL2 is better than the two QDI Excite approaches. This is inversely proportional to the number of collection selection counters.

An estimation from Fig. 6.17 and Fig. 6.18 shows a 10% decrease in collection selection performance between QDI AOL and QDI AOL2. This is contrasted by a reduction of the index size by roughly 30 percent. The difference becomes even larger when the Excite 1997 and Excite 1999 query logs are considered: compared to QDI AOL, a 75 percent size reduction is achievable at the expense of about 30% collection selection performance loss. However, there was one particular setting with informational queries – depicted in Fig. 6.20 – where the Excite 1999 query log achieved the best recall of all QDI strategies. Although this can be contributed to chance that the query log works better with the IP Split, it shows that there is room to improve collection selection while reducing the index size.

6.4 Discussing the effects of highly discriminative key parameters

There are two ways to look at the effects of HDK parameters: index size and collection selection performance. The HDK parameters are: term set frequency maximum, collection maximum and window size.

Index size

Appendix D shows the effects of the term set frequency maximum and the collection maximum on the number of term sets and the number of collection identifiers in a global index. This paragraph discusses the effects and describes the generic trends that can be concluded from those numbers.

The number of term sets depends on the term set frequency maximum. Lowering the maximum means that more keys become too frequent within a collection. This results in the generation of more multi-term keys. The number of collection identifier counters is limited by the collection maximum. This parameter limits the number of collections that are stored per term set. For our experiments, this means that the number of collection identifier counters can be at most 5, 10, 20 or 50 times as large as the number of term sets, depending on the chosen collection maximum.

Analysis of the graphs shows that the total number of term sets decreases when the term set frequency maximum is increased. The collection maximum has no effect on the number of term sets in the global index. For the number of collection identifier counters (the better estimate for the index size), the effect differs by the number of collections in the corpus splits:

100 collections The number of counters doubles when the collection maximum is increased from 5 to 50. The number of counters hardly decreases when the term set frequency maximum is increased.

11512 collections The number of counters hardly increases when the collection maximum is varied, but the number of counters decreases when the term set frequency maximum is increased.

Collection selection performance

Test results can be aggregated by plotting different graphs in one figure. Trends may emerge from the aggregated graphs when all but one HDK parameter are fixed. For example, decreasing the collection maximum improves the recall for informational queries in the Random 100 corpus split. These analyses can be done for every type of corpus split, every query type and every highly discriminative parameter type. The graphs that are required to perform this analysis have been left out of this thesis, as at least 20 graphs are required per parameter type.

Table 6.2 shows the effects of term set frequency maximum and collection maximum on the collection selection performance. The table shows how the two parameters should be altered to improve precision and recall considering the different query types and corpus splits. The Overlap3 11512 has question marks for the collection maximum. Varying the collection maximum was impossible due to memory limitations for the resulting global index. A minus sign behind a parameter denotes no influence.

	Improve precision				Improve recall			
	Navigational		Informational		Navigational		Informational	
Random 100	tf -	cm -	tf -	cm -	tf -	cm -	tf down	cm down
Random 11512	tf -	cm -	tf -	cm up	tf up	cm down	tf -	cm up
Overlap3 11512	tf down	cm ?	tf down	cm ?	tf down	cm ?	tf down	cm ?
IP Merge 100	tf up	cm down	tf -	cm -	tf up	cm down	tf up	cm -
IP Split	tf up	cm -	tf down	cm -	tf up	cm up	tf down	cm up

Table 6.2. HDK parameter effects on collection selection performance.

The table shows that it is not possible to conclude anything sensible from this information. This can be contributed to the corpus split properties. To find relations for parameters, tests would have to be repeated by generating many random corpus splits, preferably also with different numbers of collections.

Window size

Varying the window size appeared to be infeasible due to memory limitations. Increasing the window size from 6 to 9 terms results in a rapid increase of generated term sets. With a window size of 6, the worst case amount of keys added to the index is $\binom{6}{3} = 20$. In this worst case, every single term and double term set is frequent, and every triple term set is infrequent. The infrequent term sets are added to the index. When the window size is increased to 9, the worst case number increases to $\binom{9}{3} = 84$. These numbers show the potential increase in number of keys that can be generated within *each* window. With a small relatively small increase in window size, too many keys become infrequent and will be added to the index, resulting in an index that is too large. Tests showed that a window size of 6 was the limit for which most generated indices fitted in 28GB of main memory with our query-driven indexing system.

6.5 ColRank

Results

The collection selection performance of the ColRank method can be compared to other collection selection methods in Section 6.2. The graphs in that section show that ColRank is outperformed by LMDS and Sophos.

Table 6.3 shows the mean and standard deviation of the scores that were generated by standard query-driven indexing and ColRank. The standard query-driven indexing scores are stated for navigational queries and informational queries. The selected standard query-driven indexing settings (a term set frequency maximum of 500 and collection maximum of 20) were chosen arbitrarily. As ColRank is a static ranking algorithm, the scores are equal for both query types.

	QDI navigational		QDI informational		ColRank	
	Mean	σ	Mean	σ	Mean	σ
Random 100	-1.2378	0.41941	-2.4041	1.2757	-2.0000	5.7467e-13
Random 11512	-3.7285	1.0013	-5.0855	1.4178	-3.6854	0.19630
Overlap3 11512	N/A	N/A	N/A	N/A	-4.2181	0.054631
IP Merge 100	-1.4178	0.64258	-2.3757	0.90809	-2.7893	0.0014132
IP Split	-2.6728	1.3071	-3.4337	0.97330	-3.7023	0.30218

Table 6.3. Mean and standard deviation (σ) of scores for navigational and informational queries with query-driven indexing (QDI AOL tf500cm20) and for ColRank.

Fig. 6.25 shows the *total recall* graph for ColRank on the Random 100 and the IP Merge 100 corpus splits with the informational queries. The graph shows that the recall is better when ColRank is used for collection selection with the IP Merge 100 corpus split. For reference, the graph also shows the query-driven indexing performance on the IP Merge 100 corpus split.

Discussion

Collection selection performance

For the conceptual idea of ColRank to work, collections would need to have clusters with:

- relevant documents and many incoming links from high ranked collections
- non-relevant documents and less incoming links from high ranked collections.

It is hard to motivate why ColRank would work with random corpus splits, as documents are clustered randomly. This means that the links between the collections are also random, making it impossible to motivate when and why a

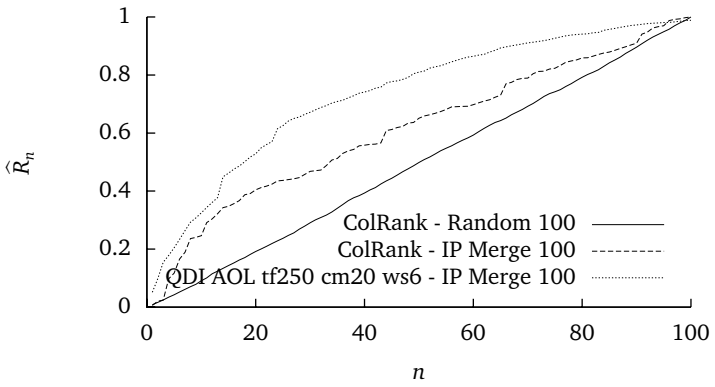

Fig. 6.25. Corpus split: *Random 100 and IP Merge 100*; query type: *informational*; graph: *total recall*.

collection should be ranked higher. This is reflected in the collection selection performance for all random corpus splits as shown in Section 6.2.1; ColRank is not able to outperform random collection selection.

To motivate the latter, we can take a look at the cumulative recall graph where ColRank seems to achieve the best recall of all random corpus splits: the Random 100 corpus split with informational queries (depicted in Fig. 6.4). The simplest way of showing why this is equal to random collection selection is by looking at the total recall graph of the same experiment; this graph was shown in Fig. 6.25. The graph shows a linear collection of merit; this is exactly what can be expected for random collection selection.

In case of splits based on IP address, we may presume that the link structure should be comparable to the Internet. On the Internet, PageRank is believed to do a good job in selecting documents. More links to a document means that it is more important. This idea makes sense on document level, but on collection level “high value” links can point to important documents, but those documents can be clustered with many unimportant documents, that also have “low value” incoming links. Following this reasoning, ColRank will work if high ranked collections mostly contain important documents, and low ranked collections mostly contain unimportant documents. Research is required to see whether the existence of important documents on web sites is correlated with the authority of a website.

Fig. 6.25 shows that ColRank does perform better on the IP Merge 100 corpus split than on the Random 100 corpus split, where its performance was on par with random collection selection. The same difference is shown between the IP Split and Random 11512 in Fig. 6.8 and Fig. 6.20. This is evidence that ColRank performs better on IP based splits. However, ColRank performs less good than all other approaches. Maybe, playing with the decay factor in the ColRank algorithm may positively affect the scores.

Collection scores

Better ranking scores is one requirement to successfully combine ColRank scores with other scores to improve ranking. Another requirement for successful score combination to improve ranking is an algorithm that is able to deal with score differences. This requires careful tuning of the scores; scores need to be in the same order of magnitude, but at the same time the differences should be large enough to really be able to make differences in ranking. When the scores are not carefully combined, the scores produced by one collection selection system will overpower the scores of the other system. In this research, this happened for all splits with 100 collections. The combined scores were different with 11,512 collection corpus splits, but did not improve collection selection.

Table 6.3 in the results above shows that the standard deviation of the scores is much smaller than the standard deviation of query-driven indexing scores. This is especially true for the corpus splits with 100 collections. This explains why the scores between ColRank and the combined scores were not different: the ColRank scores were too close to make a difference when combined with other scores.

There are two likely causes for the differences in the standard deviation of the scores:

- The link structure may actually have influenced the scores. The standard deviation is larger for the IP based corpus splits when compared to random splits with the same amount of collections.
- The differences in collection sizes may have influenced the scores. The IP based corpus splits have larger deviations in collection size. This was shown in Fig. 4.3 and Fig. 4.4. Fig. 4.3 also shows that the size difference is larger for Random 11512 than Overlap3 11512. This is also reflected in the standard deviation of the scores. Larger collections are likely to have more incoming links.

Both explanations sound plausible. Either way, more research is required to see if ColRank actually ranks collections better. The results on the IP based corpus splits motivate that an improved version of ColRank may actually improve the collection ranking scores when it is combined correctly combined with another collection selection method.

6.6 Results of different scoring formulas

This section describes the effect of different scoring formulas by assigning weights to different parts of the scoring formula. This is done by varying two parameters β and γ, as described in Section 3.3.4. That section also introduced the weight shifting ScoreFunction (Formula 3.2), of which a simplified version is given below:

$$s = \log_{10}\left(\frac{(1-\beta-\gamma)\cdot \mathrm{comp}_1 + \beta\cdot \mathrm{comp}_2 + \gamma\cdot \mathrm{comp}_3}{h_{\max}}\right)$$

Table 6.4 repeats the intended behavioral properties of the components and gives the weight assignment to the components.

Component number	**Property**	**Weight**
comp_1	Length of HDKs in which query is split	1-β-γ
comp_2	Number of query terms found in one collection	β
comp_3	Sum of HDK frequencies in one collection	γ

Table 6.4. Weight assignment to components/properties

The experiments were run with the IP Split in combination with query-driven indexing using the AOL query log. Term set frequency maximum values of 250, 500 and 750 were used. The collection maximum was fixed to 50. There were two different query types, three different evaluation metrics and 400 parameter combinations, which resulted in 7200 graphs. Obviously, most of those graphs are omitted from this thesis.

Movies were made to detect patterns in the result graphs. One movie frame is depicted in Fig. 6.26, showing the effect on precision of varying γ when n collections are selected. The figure shows the results for the informational queries with $tf_{\max} = 250$ and $\beta = 0.6$.

The collection selection performance is comparable to the results obtained with the unmodified ScoreFunction and stays constant for every combination of β and γ, except when γ is around 0.4. This behavior recurred in other frames as well; when the sum of β and γ came close to 1, precision and recall dropped in most test settings. However, in two test settings precision and recall slightly increased for informational queries, namely when $tf_{\max}$ was set to 500 or 750. The performance even stayed comparable to the unmodified ScoreFunction when β and γ were both set to 0.

These observations support the notion that the length of HDKs (the first property) is important; the effect of the first component on the score calculation using the formula above is mitigated, when the sum of β and γ is close to one. The query result lists – not included in this thesis – showed that collections containing longer HDKs are no longer ranked on top of the result list when the performance dropped.

No conclusive evidence was found whether number of query terms found in one collection is more important than the sum of the frequencies of the HDK occurrences in one collection.

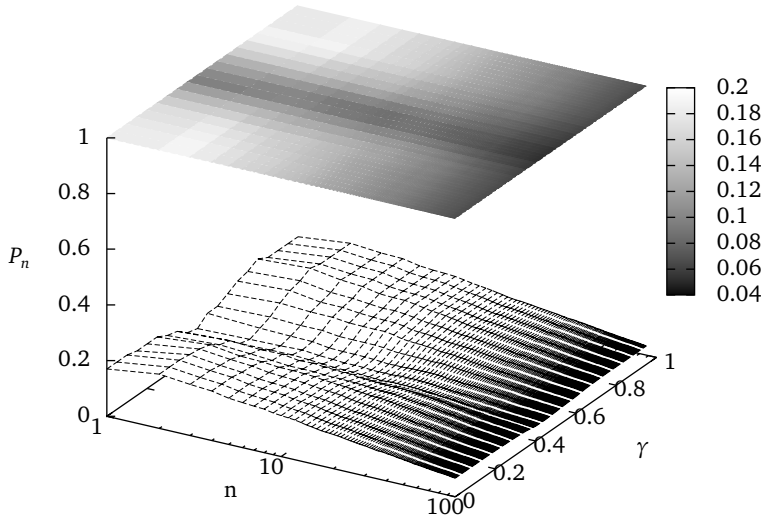

Fig. 6.26. Corpus split: *IP Split*; query type: *informational*; graph: *precision*; tf_{max}: *250*; β: *0.6*. The surface plot on top of the figure shows the same information as the graph below it: the different gradients represent different values of precision.

6.7 Difference between types of recall graphs

6.7.1 Results

Section 4.2.1 described two approaches to calculate recall. The first approach divided the collected merit with the maximum obtainable merit for the number of selected collections (cumulative recall). The second approach divided the collected merit with the total merit available in the corpus split (total recall). Fig. 6.27 shows the recall that was calculated using cumulative recall (with Formula 4.1). Fig. 6.28 shows the recall that was calculated using total recall (with Formula 4.2).

6.7.2 Discussion

The majority of all result graphs showed that total recall graphs did not reveal more detail than cumulative recall graphs. The reason is that most cumulative recall graphs start low, which means that total recall graphs will start even lower. This makes it harder to see what recall is obtained by selecting the first collections. Despite that, there are some differences between the two approaches of recall calculation, which are demonstrated by Fig. 6.27 and Fig. 6.28.

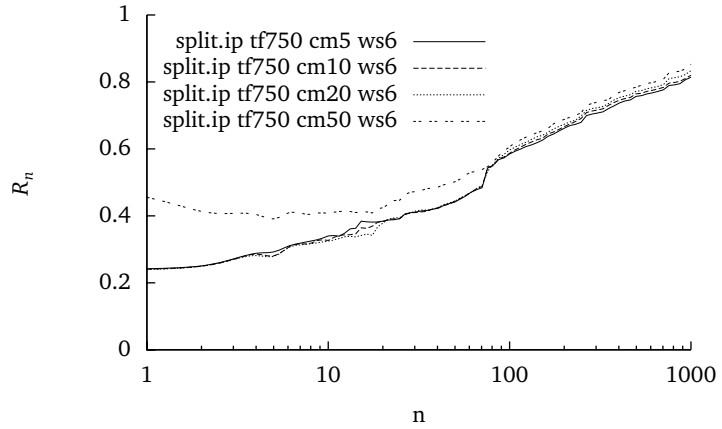

Fig. 6.27. Corpus split: *IP Split*; query type: *informational*; graph: *cumulative recall*.

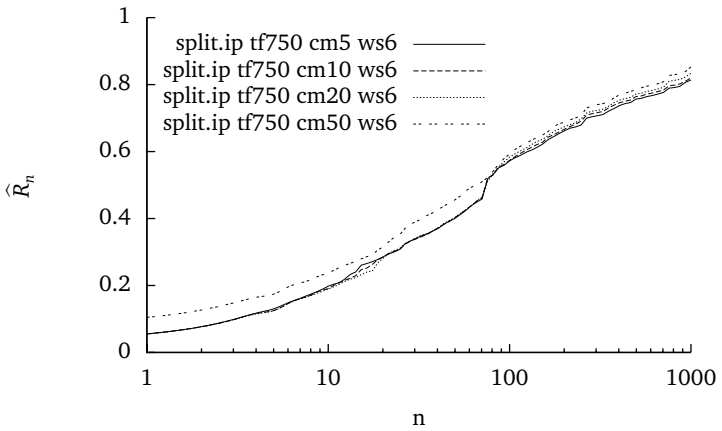

Fig. 6.28. Corpus split: *IP Split*; query type: *informational*; graph: *total recall*.

Cumulative recall graph

The cumulative recall graph from Fig. 6.27 shows that one particular configuration (tf750, cm50, ws6) performs good with at least 40% recall all the time. It may also look like much merit has been gathered from the beginning. However, this is not the case as is shown by the total recall graph in Fig. 6.28. The cumulative recall graph does not give information about the fraction of total merit collected, until the result list positions have been reached that are larger than the number of non-zero merit collections. From this point on, the cumulative recall graphs are equal to the total merit graphs.

The cumulative recall graph shows that the recall can drop. This is possible when the merit that is selected by a collection selection system is lower than the merit that could have been selected by the best possible selection system. When recall drops in a cumulative recall graph, it does not mean that no relevant collections are selected. This is shown by the total recall graph, which keeps increasing in the same area where the cumulative recall graph drops. The cumulative recall graph can only drop when the best possible retrieval system has not yet retrieved all non-zero merit collections.

Total recall graph

The total recall graphs are able to show the absolute recall, but they do not indicate how good that recall is. It can be that the retrieved recall is low according to the total recall graph, but when there are only low merit collections, the perfect collection selection system may not be able to do better than that. This information is missing in the total recall graph. This problem also exists the other way around. This was demonstrated in the discussion about ColRank in Section 6.5. There we showed that the cumulative recall graph was misleading when it came to judging whether the obtained recall was better than random collection selection. Section C.3 describes that it is hard to use the cumulative recall graph to see whether a collection selection system is better than random collection selection.

6.8 Scalability analysis

Good precision and recall is important for a collection selection system, but scalability is at least as important. This section discusses the global and local index size, the network bandwidth requirements and the expected performance in a more realistic setting.

6.8.1 Global index size

Appendix D shows the number of term sets and collection identifier counters in the global indices that were created for different corpus splits and different parameter combinations. The two numbers can help in estimating the global index size. As the IP Split is the corpus split that most closely resembles the characteristics of the Internet, we look at the number of term sets and collection identifier counters that were created for that split. Those numbers are shown in Fig. D.7 and Fig. D.8. The graphs show different parameter combinations for query-driven indexing. The best collection selection performance was obtained with a term set frequency maximum of 750 and a collection maximum of 50. The total number of collection identifier counters for this parameter combination was $6.6 \cdot 10^7$. The total number of term sets was $5.9 \cdot 10^6$. With these numbers, a minimal global index size can be calculated.

A collection identifier counter consists of a collection identifier and a counter that counts the number of term set occurrences. There are 11,512 collections in the IP Split, which is more than 2^8 (1 byte), but less than 2^{16} (2 bytes); hence a collection identifier needs 2 bytes. The counter also needs 2 bytes as the term set frequency maximum is 750. As a result, every collection identifier counter needs 4 bytes. Every term is identified by 4 bytes. When the maximum number of terms in a key is set to 3, every term set needs at most 12 bytes. Only looking at the term sets and collection identifier counters, the worst case size of the data in the global index is:

$$(5.9 \cdot 10^6 \cdot 12\text{bytes}) + (6.6 \cdot 10^7 \cdot 4\text{bytes}) = 320\text{MB}$$

Unfortunately, storing and retrieving the data items does not come for free. Our Java implementation used a separate index to store the mapping of a real term to a 3 byte term identifier (which was internally stored using 4 bytes). This structure is **50MB**. To speed up search, our implementation used the multi-level term index, as described in Section 5.2. This first level in this structure is an array that stores pointers for all possible single terms. There is room for $1.6 \cdot 10^7$ unique terms. Every pointer uses 8 bytes, as we used a 64-bit machine. This first level pointer array uses **16MB**. Whenever a term is present, a pointer points to a node of a tree. This node contains two Java TreeMap objects: one storing all next level terms (for the multi-term keys) and one storing the collection identifier counters. The number of nodes is equal to the number of term sets. A node is estimated to be 60 bytes (this is a rough estimate). All nodes require **335MB** ($= 5.9 \cdot 10^6 \cdot 60$bytes). Finally, we also need a pointer (8 bytes) for every collection identifier counter, which takes up **500MB**.

The sum of the data structure space requirement is 900MB. The total global index space requirement is estimated to be 1.2GB. The corpus size is 10GB, which is reduced to 3GB after cleaning, stop word removal and word stemming. The actual index data of the processed split was estimated to be 320MB.

6.8.2 Local index size

The local index size is dependent on the collection properties:

- Large collections are likely to generate more keys.
- Many term sets will be generated if the text contains many different words in a window size.

The local index is almost the same as the global index, but it does not have to store a collection identifier; it only needs to store counters for every term set. However, it does need to store more term sets, because the local index needs to keep track of the term sets that are too frequent.

The combination of these characteristics makes it hard to theoretically estimate the local index size. Experiments showed that the local indices were small for the Random 11512 corpus split with its many small collections, but were larger than the global index for the collections in the 100 collection corpus splits and for the large collections in the IP Split.

6.8.3 Network traffic requirements

Two major communication cost components are involved to create a global index at a broker that can answer the users' queries:

- The query log is required at the site that wants to build a local index. The log is used to reduce the number of generated term sets. The biggest query log that we used in our implementation was the 175MB AOL query log (without additional metadata). That log can be compressed to 35MB. When the log only contains queries that occur at least two times in the original query log, the log size can be reduced to 75MB (15MB compressed).
- In our test scenario, the local indices could become bigger than the actual global index. Not all of the information in the local index will have to be sent to the broker. Only the keys that are locally infrequent will be sent to the broker. We assume that one collection contains no more than half the amount of unique terms that exist in all collections. We also assume that the amount of local index data to send to the broker is at most half the size of the global index. As the IP Split was a realistic view on the Internet, we can roughly estimate that sending the local index to the broker requires at most 160MB for large sites. This estimated 160MB is just numbers, so we can expect compression to reduce this number by at least 50%.

The estimated network traffic is required for every participating site. Once the information is sent to the appropriate places, the communication cost is much smaller than traditional web search systems when it comes to keeping the index up to date. Query log updates can be sent incremental. The same approach can be used for changes to the local index. Changes to the local index can also be sent in batches. The big advantages of local index updates in our scenario is that they can be done as soon as the local index has been updated to reflect the changes in the site(s). This results in the freshest global index possible.

6.8.4 Using a larger test corpus

Using Heaps' law, this section theoretically analyses the Sophos index effects if the test corpus size is increased from 10GB to 1TB. Heaps' law, $V = K \cdot n^{\beta}$, is an empirical law that predicts the number of distinct words – the vocabulary V – given a corpus with n words [32]. The formula has two parameters K and β, that depend on the text. For English texts, K varies from 10 to 100 and β varies from 0.4 to 0.6.

For web data, β typically is larger due to the existence of multiple languages and typing errors [4]. We assume $\beta = 0.6$, which gives $K = 14.5$ for the WT10g corpus ($n = 500,000,000$ and $V = 2,400,000$). Assuming that K and β remain constant with increasing corpus size, the number of unique terms in a 1TB collection can be estimated: $V = 14.5 \cdot (500,000,000 \cdot (1\text{TB}/10\text{GB}))^{0.6} \approx 38,000,000$. This is about 16 times more unique terms.

As 38,000,000 possibilities can be represented using 26 bits, the unique term index can assign a unique term identifier to every term using 26 bits. This can be stored using 4 byte integers, which is the same amount as our prototype was working with. Assuming that the new terms have the same overlap ratio, we estimate a unique term index increase with a factor 16, resulting in an 800MB unique term index at the broker.

The local index sizes will remain relatively constant with a larger corpus. Of course, there will be a small amount of collections that are significantly larger than the current largest collection, but the average collection size will be the same. This contrasts with the global index size, which will become larger.

The global index uses the multi-level term index to store term sets and collection identifier counters. To deal with the larger amount of terms, the multi-level term index array can be increased to store 48,000,000 unique terms. This increase the array size from 16MB to 48MB. This is neglectable compared to the space requirement for new term sets and collection identifier counters. It is hard to make a good estimate of the new term sets that will be created. Using Zipf's law, we can predict that the number of very frequent term sets will increase. The additional storage required for these very frequent keys is bounded by the collection maximum, which limits the number of additional collection identifier counters. However, the long tail of infrequent keys will increase as well. These keys cause an increase in space required for the global index. Whether the two will level out is a subject for further research. If we assume that they do, the global index size increases by a factor 16, which means that is will be 19GB.

Concluding, we estimated the space requirement at the broker to be 1.2GB for the 10GB WT10g (roughly 10%). A 1TB corpus is estimated to have a 20GB space requirement at the broker (5%).

6.9 Summary

This chapter described many effects of the collection selection experiments that were performed with LMDS and Sophos. The corpus splits and query types significantly affect collection selection performance; Sophos performed best with informational queries in the most realistic collection split resembling the web, but had an overall lower collection selection performance in other situations when compared to LMDS. Query-driven indexing significantly reduced the index size up to 90% with the Excite 99 query log, with an estimated loss in performance of about 30%. ColRank did not outperform LMDS and Sophos, but it did appear to perform better than random collection selection on IP based splits.

Varying the different HDK indexing parameters revealed no univocal relation for collection selection performance. The number of term sets decreased when the term set frequency maximum was increased. There is thin evidence that the number of collection identifier counters is inversely related to the number of collections and the collection maximum. There may also be an inverse relationship

between the number of collection identifier counters and the number of collections with the term set frequency maximum.

Tests with the weight shiftable ScoreFunction supported the notion that the key length – of the keys that are used to find the collections in the index – is the most important ranking attribute for collection ranking.

This chapter also described the difference between cumulative and total recall graphs. The cumulative recall graph reveals the most detail when it comes to comparing recall of collection selection systems. However, the total recall graph is required to show the absolute performance in terms of recall.

The scalability of Sophos was theoretically analyzed, showing an estimated index space requirement of 10% of a 10GB corpus, which drops to 5% for a 1TB corpus. A more detailed estimate of the space and traffic requirements showed that the broker will end up having 320MB of term sets and collection identifier counters with a 50MB unique term index. The estimated space requirement at the broker is 1.2GB with the index structure required to store the actual data. The peers will have a larger local index, of which the largest peers need to send an estimated 160MB. They also need a query log which can be compressed to 15MB.

7

Conclusion

There's two possible outcomes: if the result confirms the hypothesis, then you've made a discovery. If the result is contrary to the hypothesis, then you've made a discovery.

– Enrico Fermi

This research compared the following collection selection methods:

- Language modeling with Dirichlet smoothing (LMDS)
- HDKs
- HDKs with query-driven indexing
- A PageRank-like algorithm to rank collections (ColRank)
- HDKs with query-driven indexing in combination with ColRank.

The first method acted as a baseline to see whether HDKs (in combination with other approaches) would be a viable approach to select collections. The WT10g corpus was used to assess the performance of the five methods. The WT10g corpus is a crawl from the Internet and has been split in five different ways. Three of them were random splits and two of them were based on IP address. The selection performance was measured in terms of precision and recall using queries that were issued on WT10g corpus splits. Two types of queries were used: navigational (home-page finding) and informational (ad-hoc). This chapter describes the conclusions that can be drawn from the results of the experiments.

7.1 Corpus splits and query types

Research question 1: What is the effect of splitting the document corpus?

Although corpus splits were made once per type of split, e.g. no n-fold cross validation, we can still compare collection selection systems with each other, because we used 145 navigational and 50 informational queries on every corpus split. The choice of corpus split and query type influenced the precision and recall of the collection selection systems. The differences produced by the those two test properties were consistent for all collection selection methods. For example, the precision degraded for all collection selection methods when random corpus splits were used instead of IP based corpus splits. The effects of these properties are described and analyzed in Sect.6.2. We can draw two conclusions from changing the corpus splits and the query types.

1. The combination of random corpus splits and navigational queries proved to be difficult for collections selection methods based on HDKs. LMDS showed that it is possible to achieve precision and recall figures that are better than random collection selection in that scenario.
   As random corpus splits have the (relevant) documents randomly divided over all collections, the collections do not have many relevant documents. The navigational queries have a low amount of relevant documents. This is evidence for the conclusion that HDKs is not suited for finding small pieces of relevant information in the corpus.
2. On the Internet, every web site – running on a certain IP address – contains many documents (web pages). The IP Split is a corpus split where the web site based document partitioning based remained intact, i.e. every collection in that split contains the documents that came from one IP address. This was our most realistic test scenario.
   HDKs with query-driven indexing outperformed LMDS when it came to informational queries. The informational queries have relatively many relevant documents. In the IP Split, relevant documents are merged in collections (as shown in Appendix B). This is evidence for the conclusion that HDKs with query-driven indexing is suited for selecting collections that contain many relevant documents with respect to one query.

The Overlap3 11512 corpus split – the split where every WT10g document was randomly assigned to three of 11,512 collections – was a huge corpus split. HDKs with query-driven indexing was only capable of running tests when a query log was used to prune keys that occurred less than two times in the query log. Even then, query-driven indexing was only able to finish two out of twelve tests; all other tests ran out of 28GB of memory. Large splits are useful to analyze the scalability of a collection selection system. LMDS was able to run all tests.

7.2 Highly discriminative keys

Research question 2: What factors are important to assign the highest score to the collection with the largest amount of relevant documents?

A highly discriminative key is a term set that occurs in a limited number of collections, thereby acting as a discriminator to select those collections. The keys are generated by looking at all possible combinations of terms within a particular *window size* of terms in a document. This window size is moved from the beginning to the end of the document. The maximum window size was 6. A larger window size resulted in too many possible combinations making the system run out of memory.

If a term set occurs more times than the *term set frequency maximum*, it is called frequent key. Every infrequent key is sent to a global index. In the global index, a *collection maximum* defines the number of collections (and corresponding frequency counter) that is stored per key. The term set frequency maximum

and collection maximum were the two parameters that were varied to see the effect on collection selection performance and index size. Our tests did not reveal a trend for the collection selection performance when varying the two parameters. Section 6.4 discusses the effects on the global index size. Depending on the number of collections in a corpus split, the effect on the global index size could be summarized as follows:

100 collections The index grows when the collection maximum is increased and shrinks when the term set frequency maximum is increased.

11512 collections The collection maximum hardly influences the index size. The index shrinks when the term set frequency maximum is increased.

The global index stores term sets of one, two or three terms. Every stored term set has a maximum number of collections with an associated term set frequency within that collection. When a user has a query of three terms, the global index may contain a key with all query terms. In that case, the length of the key used to select collections is three. But it may also use three single term keys. Using an intersection of the collection list, it can also select collections that contain all three query terms. The length of the keys is one in that case. It is important to mention that our system did not look for keys with a smaller length if one key was found.

The length of keys used to select a collection, the number of query terms found in a collection and the term set frequency for the keys within a collection are three properties that were used in a formula that creates scores to rank the collection. Our test results, described in Section 6.6, provided evidence that the length of the key is the most important variable to select collections. No conclusive evidence was found about the importance of the other two properties.

7.3 Query-driven indexing

Research question 3: What is the effect of pruning highly discriminative key indices using query logs?

Query-driven indexing can be applied on top of the HDK paradigm to reduce the global index size. Section 6.3 shows that the index size can be reduced by 70% when all keys are removed that do not occur in the AOL query log. Further index size reductions are possible by application of other strategies, such as pruning keys that occur less than two times in the AOL query log. Other query logs, such as the Excite query logs, can also decrease the index size (75% compared to standard query-driven indexing). The general trend was that a reduced global index size resulted in degraded precision and recall. There was one exception with the Excite query log, which gave the best results of all collection selection systems. This proves that strong index size reduction does not have to affect collection selection performance.

The latter conclusion is important, as indexing with HDKs without any form of index size reduction is infeasible. Only a small selection of tests with the Random 11512 corpus split was able to run without query-driven indexing. The rest of the tests, with all other corpus splits, were unable to run due to memory limitations.

7.4 ColRank

Research question 4: What is the effect of using distributed collection-level "PageRank" scores of peers to rank and select them?

The calculation of Google PageRank scores can be distributed. The same approach can be used to distribute the calculation of scores for collections. As this should not affect the scores, a centralized version of a collection ranking algorithm has been implemented and tested. This algorithm is called ColRank. There is no test where ColRank – or the combination of ColRank and HDKs with query-driven indexing – is able to outperform any other approach. ColRank performs better than random collection selection on IP based corpus splits, but it is not as good as any other collection selection method. Changing the decay factor in the ColRank algorithm may improve the collection ranking.

The results on IP based corpus splits do motivate further research. However, it will also be interesting to see whether ColRank will outperform size-based ranking of collections or a simple link-based algorithm such as InDegree which counts the number of incoming links in a collection.

7.5 Scalability

There are three types of queries on the Internet: navigational (find a home page), transactional (buying objects) and informational (ad-hoc search). We created a corpus split that closely resembled the Internet. We have shown that HDKs with query-driven indexing was able to outperform a centralized information retrieval system in one particular scenario. The results was observed with informational queries on an Internet-like corpus split. This supports the idea that distributed web search can benefit from HDKs with query-driven indexing in terms of collection selection performance (precision and recall).

We have also shown that it is vital to apply query-driven indexing to the concept of HDKs. Not applying this technique results in too many generated term sets, leading to an unscalable solution. Query-driven indexing can reduce the global index size with 80%. We showed that the Excite query log could further reduce the index size by another 75%. We found weak evidence that ranking collections using an algorithm like PageRank may work at the web site level.

HDKs with query-driven indexing needs more work in reducing the physical size of the local and global indices. Our prototype ran out of memory while index-

ing the 9GB Overlap3 11512 corpus split on a machine with 28GB of main memory. The prototype also ran out of memory when the window size was increased or when no query-driven indexing was applied. The most common query-driven indexing tests on the 3GB corpus splits would also run out of memory when the memory limit was set to 25GB. We can give several remarks about this undesirable situation:

- The global index data itself is estimated to be less than 5% of the corpus size (320MB on a 3GB corpus split). The data structure to store this data was estimated to be almost 3 times as large (900MB). Possibly, the Java implementation on a 64-bit machine introduced extra overhead for the estimated data storage requirements.
- The entire indexing and querying was done in main memory; nothing was written to disk. This increased indexing and querying speed. Local and global indexing was typically performed in three hours. Writing intermediate results to disk would reduce main memory requirements at the expense of indexing speed.
- Everything was done on one machine. The machine stored multiple large local indices and a global index in main memory. Distributing this to multiple machine would significantly reduce main memory usage on every machine.

We can conclude that there still is room for improvement on the memory usage of our prototype.

The initial upload traffic for a site participating in the distributed web search scenario with HDKs and query-driven indexing is less than with the traditional centralized web search approach. The local index data can be smaller than the size of a web site, which would have to be crawled (uploaded) in the traditional approach. The approach does require an initial estimated download of 15–35MB for the query log. Any updates to the local index and query log can be done incremental.

7.6 Summary

In which ways can web search be improved with collection selection from distributed information retrieval using a combination of highly discriminative keys and 1) query-driven indexing, 2) distributed PageRank?

We have shown that HDKs can be used for collection selection. It is desirable to combine it with query-driven indexing for a substantial index size reduction. Index pruning usually decreases collection selection performance, but this was not always the case. The essence of index pruning is to keep the longest HDKs that are most queried for. A good query log helps in the bootstrap problem of creating good initial local indices.

We have also shown that the combination of HDKs with query-driven indexing is suitable for answering informational queries on IP based splits. We have

seen that it is much harder for the combination to create good indices of random collections, especially when tested with navigational queries. This can be contributed to the large diversity of documents (and topics) in those collections and the low number of relevant documents. The majority of web pages on the Internet will not look like random collections, but HDKs and query-driven indexing should be able to cope with such collections when globally operating search engines are taking part in the distributed web search scenario. Globally operating search engines are not limited to indexing their own site; they can index sites of a particular domain (in which we may get collections like the IP Merge 100) or they can have the aim of finding the best web sites for a large portion of queries. This demonstrates the significance of testing with different types of corpus splits and query types.

Finally, we did not find conclusive evidence that ColRank does not work. More research is required to see if ColRank differs from size based ranking or InDegree. The ColRank scores should improve as well: relevant collections should get a higher score with more deviation from other scores. The scores will also have to be matched better with scores from other collection selection systems to positively influence ranking.

8

Future work

To make no mistakes is not in the power of man; but from their errors and mistakes the wise and good learn wisdom for the future.

– Plutarch (46AD–120AD)

Collection selection using HDKs seems to be a promising approach, which motivates further research. This chapter sketches some possible research directions focusing on increasing the effectiveness of the method in a test setting of larger scale.

8.1 Implementing an optimized distributed system

It took about three hours to build the local and global indices for most corpus splits. An intermediate global index acted as a buffer to batch process index additions to the real global index. This was desirable, because inserting index entries one by one slowed down as the global index size increased. All tests ran on one machine, where the unique term index was built only once and where it was used for all indices.

A possible approach to drastically decrease the index generation time is to distribute the index generation using many machines, thereby simulating a more realistic environment. In Sophos, global index generation took three hours for up to 11,512 collections, so one local index can be constructed in just under a second. Even if we construct the unique term index for every collection, one collection could be indexed in under 5 seconds. The load on the global indices can be divided by splitting the global index in multiple parts, where each part is responsible for a set of keys.

This approach exploits parallelization and lowers memory requirements per machine, which makes it possible to test more different indexing settings in significantly less time. It will also enable testing with larger IP based corpora, which is required to see in practice how collection selection using HDKs scales.

8.2 Improving collection selection results

Several approaches can be investigated to improve the collection selection results:

Collection-dependent parameter tuning: Appendix A showed that the collection sizes varied from hundreds of bytes to hundreds of megabytes. It may be a better idea to do collection-dependent parameter tuning, because smaller collections are likely to have less frequent keys.

Improved scoring algorithm: There is still room to improve the scoring algorithm. One way is to give independent weight factors to every scoring component that was introduced in Section 3.3. The ColRank scores can be included as another scoring component. The optimal weight mixture can be discovered by experimentation.

When a query only consists of locally frequent keys, our solution is able to retrieve collections containing the query terms by finding multi-term keys that contain those query terms. This approach will not give the best possible results. It may be a better idea to allow some locally frequent keys into the global index. The amount of keys can be related to the collection size. Using only local information it is hard to decide which frequent keys are globally rare by only using local collection knowledge. Possibly the following approaches help in finding locally frequent keys that are discriminating with respect to other collections:

- After detection of collection language, a Part-of-Speech tagger for that language could be used to detect nouns and verbs. Frequent keys are only allowed to be sent to the broker when they contain nouns and/or verbs.
- A peer can use language modeling to build a language model of its own collection. This can be used to model the probability distribution of term set occurrences. A 'universal corpus' language model can be constructed by analyzing a huge amount of web text. This model can be sent to every peer. The peer can use its own language model and the universal language model to discover which term sets occur more frequent in comparison to the universal corpus. Those term sets are candidates to send to the broker.

Queries can help in identifying locally frequent keys that are interesting to add to the index, as explained in the next section.

8.3 Query-driven indexing

Sophos used the query log as a filter to discard keys not occurring in the query log. After creation of the reduced global index, Sophos was tested using evaluation queries. This approach gave a good idea about the possible index size reduction and loss in collection selection performance, but also showed that the loss was relatively large compared to the results with the non-pruned index. This was caused by the removal of several evaluation query terms. It demonstrates the need for query log updates at the local peers to deal with new unseen terms. It would be interesting to use another indexing strategy, based on the popularity of queries as described in Section 2.3.3.

When a new peer joins in the distributed web search system, it starts by downloading the latest query log summary containing the most popular queries.

After local key generation, the query log can act as a filter to send popular keys to the broker. With the use of such a query log to filter keys, it may become feasible to also send locally frequent keys to the broker. Once the initial bootstrapping has been completed, the peer receives (a few) queries. Based on the queries that it receives, it can decide to activate new keys that are sent to the broker. The broker can also opt to send a popular query to presumed promising collections that are not yet represented in the global index. When those collections contain relevant documents, they will generate a result list that is sent to the broker and they may active corresponding keys. This approach will gradually improve recall and precision when more users pose queries.

A

Zipf in corpus splits based on IP address

The graphs in this section show the document distribution for the IP Split (Fig. A.1) and the IP Merge 100 corpus split (Fig. A.2). The x-axis shows the ranked order of the collections by number of documents. The graph is scatterplotted with both axes plotted logarithmic. The collection size distribution is plotted in the same way for the IP Split (in Fig. A.3) and the IP Merge 100 corpus split (in Fig. A.4).

In case a scatterplotted log-log graph can be estimated by straight line, the data conforms to Zipf's law. The IP Merge 100 graphs (Fig. A.2 and Fig. A.4) can be estimated with straight lines when the last collection is left out (the last collection consists of 88 collections, instead of 116 collections). The document distribution for the IP Split can be estimated by a straight line for collection 100–11512. The size distribution is a bit harder to estimate by a straight line: the 50^{th} up to the 7000^{th} collection can roughly be estimated by a straight line.

Possible explanations for the decreased ability to estimate the IP Split graphs with a straight line are that the corpus is not large enough and that the long tail is not existing as the original corpus has an artificial minimum of 5 documents per server.

Having said that, the graphs do show that the IP based splits are reasonably conforming to Zipf's law.

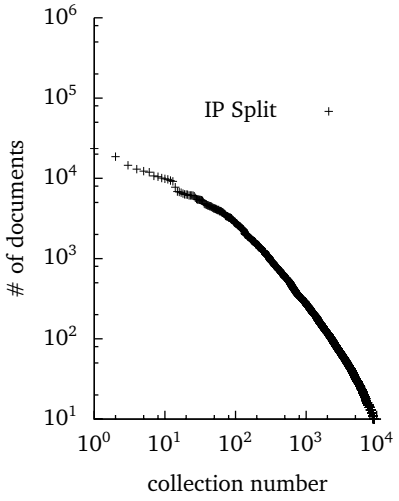

Fig. A.1. Document distribution for the IP Split (11,512 collections)

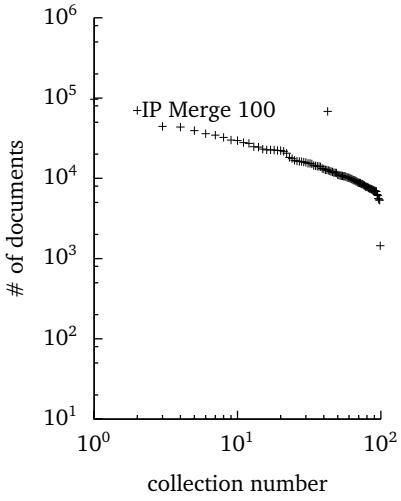

Fig. A.2. Document distribution for IP Merge 100

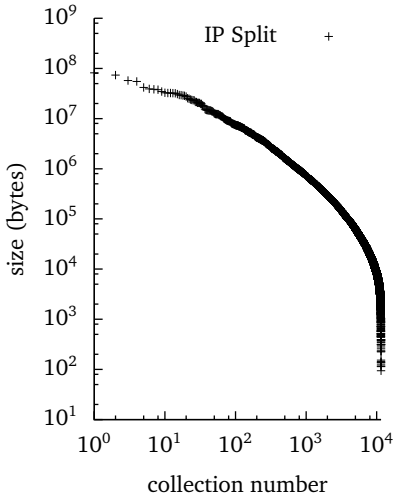

Fig. A.3. Collection size distribution (in bytes) for the IP Split (11,512 collections)

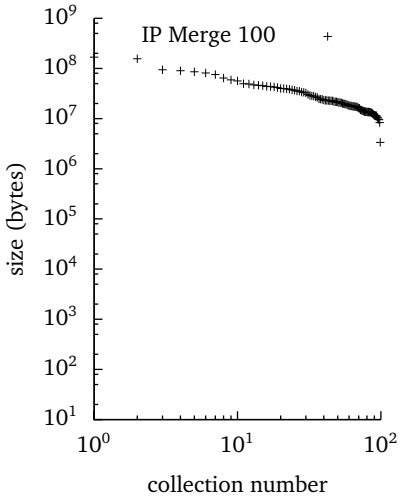

Fig. A.4. Collection size distribution (in bytes) for IP Merge 100

B

Merit in corpus splits

Fig. B.1 shows the amount of merit in the collections from the corpus splits that consist of 11,512 collections. This merit is calculated for the navigational queries. It is calculated by summing the merit per collection for all 145 queries. The collections are then sorted descending by amount of merit. Note that the axes are plotted logarithmic. This makes is it harder to see how many collections have non-zero merit. This information is displayed more clear in Table B.1. This table shows the amount of non-zero collections for each of the corpus splits and query types. Fig. B.2 shows the merit for the 50 informational queries for the corpus splits with 11,512 collections. Similar information is displayed in Fig. B.3 and Fig. B.4 for the corpus splits with 100 collections.

The most obvious difference between the corpus splits based on IP address and the random corpus splits is that the merit deviation is larger for splits based on IP address; the IP Split and IP Merge 100 have some collections that contain a large amount of merit.

Corpus split	**#non-zero merit collections (navigational queries)**	**#non-zero merit collections (informational queries)**
IP Split	169	900
Random 11512	251	2851
Overlap3 11512	731	6601
IP Merge 100	76	100
Random 100	91	100

Table B.1. Non-zero merit collections within corpus splits

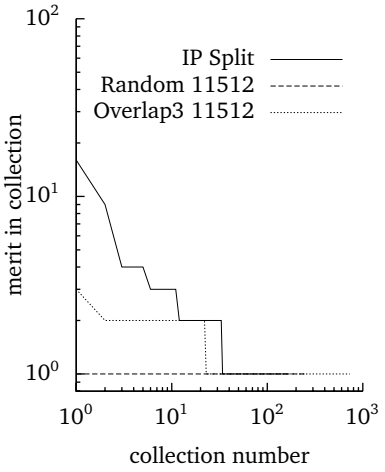

Fig. B.1. Cumulative merit distribution in corpus splits with 11,512 collections - navigational queries

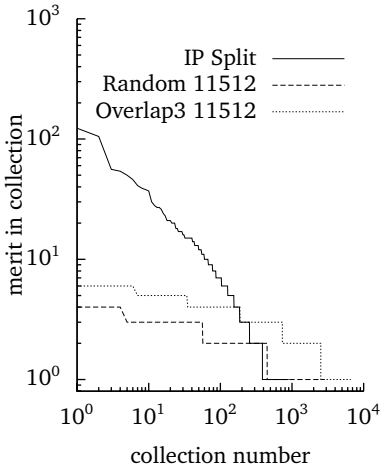

Fig. B.2. Cumulative merit distribution in corpus splits with 11,512 collections - informational queries

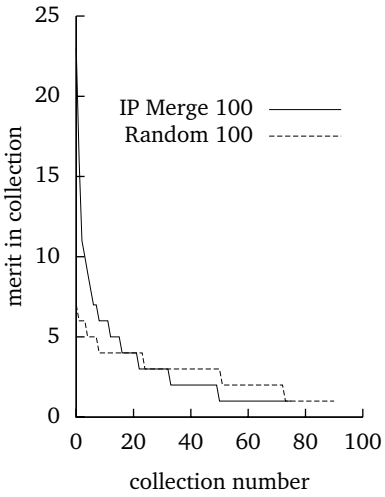

Fig. B.3. Cumulative merit distribution in corpus splits with 100 collections - navigational queries

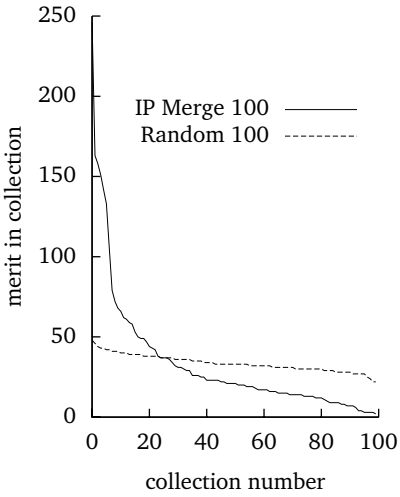

Fig. B.4. Cumulative merit distribution in corpus splits with 100 collections - informational queries

C

Random collection selection performance

This appendix explains the expected collection selection performance when collections are selected randomly. The collection selection performance will be described in terms of precision and cumulative recall. The explanation uses an imaginary test corpus split to make the expected performance more insightful. The corpus split properties are:

- 100 collections
- 20 relevant collections (collection with non-zero merit)
- 5 collections with high merit (a merit value of 4)
- 15 collections with low merit (a merit value of 1)
- total merit value of 35.

This appendix concludes with an analysis on the recall performance of ColRank on the Random 100 corpus split.

C.1 Precision

The hypergeometric distribution is a discrete probability distribution that can be used to model the expected precision. The distribution can be used to calculate the probability of selecting a collection with merit in a sequence of n selections from a finite population of 100 collections without replacement. It can also be used to calculate the expected value of the number of selected collections with merit after selecting n collections.

The formula for the expected value in a hypergeometric distribution is $n\frac{R}{N}$, where n is the number of selected collections, R is the total number of collections with merit and N is the total number of collections. The fraction of that expected value and n is the expected precision, which is a constant: $\frac{R}{N}$ The expected precision for every n in the test corpus split is $\frac{20}{100} = 0.2$. This is shown in Fig. C.1.

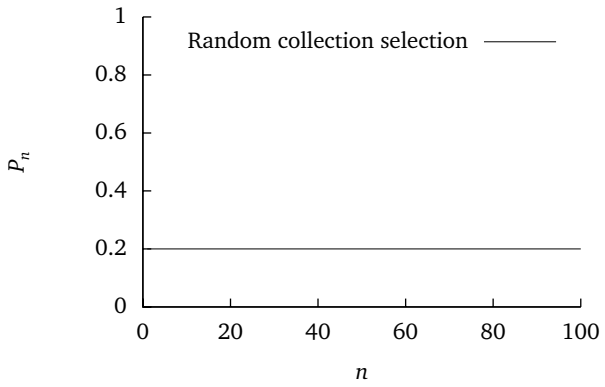

Fig. C.1. Expected precision for random collection selection.

C.2 Cumulative recall

In case of random collection selection, we define the cumulative recall after selection of n collections as the fraction of merit collected by a random collection selection system and the merit collected by the best possible selection system. The expected merit for a random collection system will have to be calculated before the cumulative recall can be calculated.

This can be done using a multivariate hypergeometric distribution. The expected collected merit after selection of n collections is defined as:

$$E(X_n) = \sum_{i}^{m_{\max}} \frac{n m_i i}{N} \tag{C.1}$$

In this formula, m_i is the number of collections with a merit value of i and $m_{\max}$ is the maximum merit value that is present within a collection. N is the total number of collections. The expected merit for the test corpus split is shown in Fig. C.2. That figure also shows the collection of merit by the best possible collection selection system.

Fig. C.3 depicts the fraction of the two merit graphs; it shows the expected recall for the test corpus split. The graph is constant in the beginning, because the collection selection systems have a constant increase in selected merit. The high merit collections all have the same merit value of 4. The best possible ranking system has a constant increase in selected merit for the first five collections. The random collection selection system also has a constant increase in selected merit (of 0.35 per collection). This results in a constant fraction. The rate of merit accumulation will drop for the best possible collection select system as soon as all highest merit collections have been selected. From that moment, the recall graph shows a rise. It will increase linear once the best possible collection system selected all relevant collections.

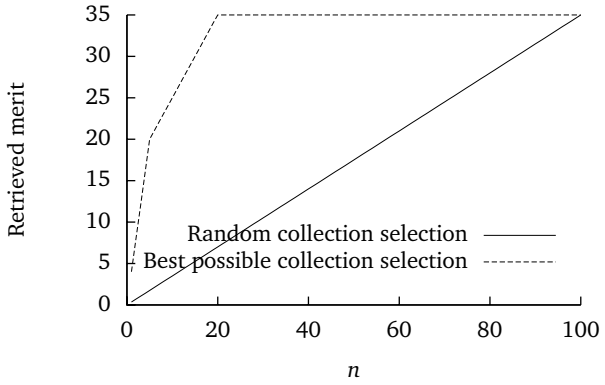

Fig. C.2. Comparison of merit collection by best possible collection system and random collection selection system.

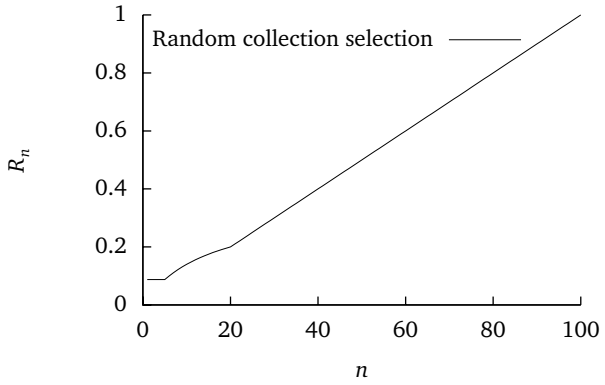

Fig. C.3. Expected recall for random collection selection.

C.3 Detecting random selection in a cumulative recall graph

The rest of this appendix analyses the cumulative recall of the ColRank collection selection method, which was used with the Random 100 corpus split with 50 informational queries. The cumulative recall graph is shown in Fig.6.4. We will show that it is hard to use the cumulative recall graph to detect whether a collection selection system is doing better than random collection selection.

On average, there are 19 relevant collections in the Random 100 corpus split for the informational queries; the number of non-zero merit collection, n^*, is 19. Using Fig. B.4 in Appendix B, we can derive that the average merit is 35 per collection in the Random 100 corpus split. This merit value is the sum of the number of relevant documents for all queries in one collection. There are 100 collections and 50 informational queries, so the average total merit per query

is $35 * 100/50 = 70$. The merit per collection per query is 0.7. This is the expected amount of merit collected per collection when collections are selected using random collection selection, denoted by $m_{\text{exp-random}}$. The best possible ranking system can select a merit of 70 within 19 collections: an average 3.7 merit per collection. This is denoted by $m_{\text{exp-best}}$.

We can now define the cumulative recall for random collection selection as the fraction between the expected collected merit and the best possible amount of collected merit at the selection of n collections. In a formula, this becomes:

$$R_{n,\text{random}} = \frac{n \cdot m_{\text{exp-random}}}{\sum_{n=1}^{n^*} m_{\text{exp-best}}} \tag{C.2}$$

Note that this formula uses average merit values to calculate $m_{\text{exp-random}}$ and $m_{\text{exp-best}}$. This is not realistic, as the merit in collections is not uniformly distributed. If a list of merit values per collection per query for the different collections is known, we can substitute $m_{\text{exp-random}}$ for Formula C.1 and use the list to calculate the expected values. We can also substitute $m_{\text{exp-best}}$ for the sum of the top n merits of the ordered collection merit list. As such a list will not positively influence the cumulative recall scores for random collection selection, we will use Formula C.2 to calculate the cumulative recall for random collection selection. The resulting graph has been plotted in Fig.C.4. For easy comparison, this figure also shows the cumulative recall graph of ColRank.

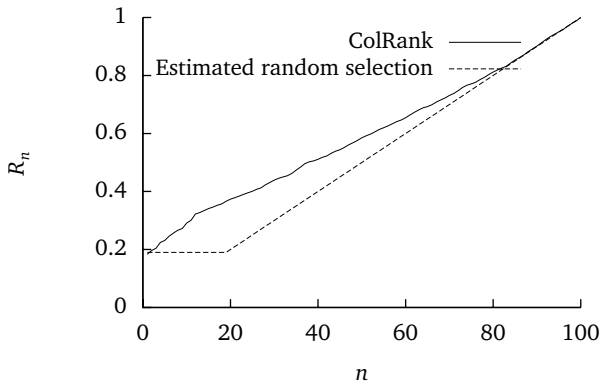

Fig. C.4. Estimated random collection selection versus ColRank on Random 100 with informational queries.

The figure shows that both graphs are not equal. One reason is that the estimated random collection selection graph has been calculated with the formula that uses average values for the merits. Earlier, we stated that this would not positively influence the scores, so this means that the difference would only be bigger. We think the difference is caused by the use of one average number of

relevant collections per query (19). In practice, one query can have many more or much less relevant collections. Using the average value for the number of collections can negatively influence the scores.

The lesson to be learnt from this appendix: using a cumulative recall graph it is impossible to see whether a collection selection system is performing better than random collection selection. Section 6.5 showed that ColRank had the same total recall graph that could be expected from random collection selection, but this section fails to theoretically reproduce that observation for cumulative recall.

D

Index size with QDI

Fig. D.1 up to Fig. D.8 show how the global index size varies by changing the HDK indexing parameters. The effects are displayed for different types of corpus splits. The graphs show two ways to look at the global index size: the number of term sets and the number of collection identifier counters (ColID counters). Every graph shows those number of occurrences for single terms, double terms and triple terms. The numbers are grouped per parameter combination. The numbers are plotted against logarithmic axes.

We can look at each graph to discover the trend that is shown by it. For example, Fig. D.2 shows the number of collection identifier counters for the Random 100 corpus split. The graph shows a small decrease in number of collection identifier counters when the term set frequency is increased. It also shows a medium increase in numbers when the collection maximum is increased.

The trends that can be spotted by this analysis have been summarized in Table D.1. The trends are grouped by corpus split. The table shows seven effects on the number of term sets and collection identifiers, ranging from strong decrease to strong increase: $---$, $--$, $-$, 0 (no effect), $+$, $++$, $+++$. The trend analysis is subjective and coarse grained, but this is no problem as we are trying to discover relationships between the parameters and the corpus splits.

	Random 100		Random 11512		IP Merge 100		IP Split	
	tf up	cm up	tf up	cm up	tf up	cm up	tf up	cm up
# term sets	$-$	0	$--$	0	$-$	0	$-$	0
# ColID counters	$-$	$++$	$--$	0	$-$	$++$	$---$	$+$

Table D.1. Effects of term set frequency maximum and collection maximum on number of term sets and collection identifier counters with different corpus splits.

The table clearly shows that the number of term sets decreases when the term set frequency is increased. The number of term sets remains unchanged when the collection maximum is increased. The table also shows a symmetry relation

for the number of collection identifier counters for corpus splits with a similar amount of collections. For example, the number of collection identifier counters shows a small decrease when the term set frequency is increased for corpus splits with 100 collections.

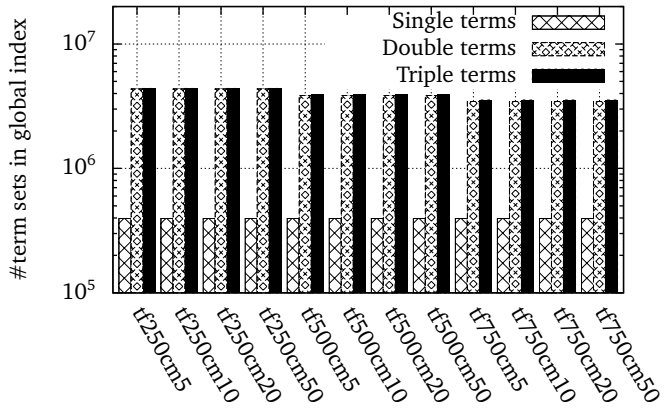

Fig. D.1. Number of term sets with Random 100.

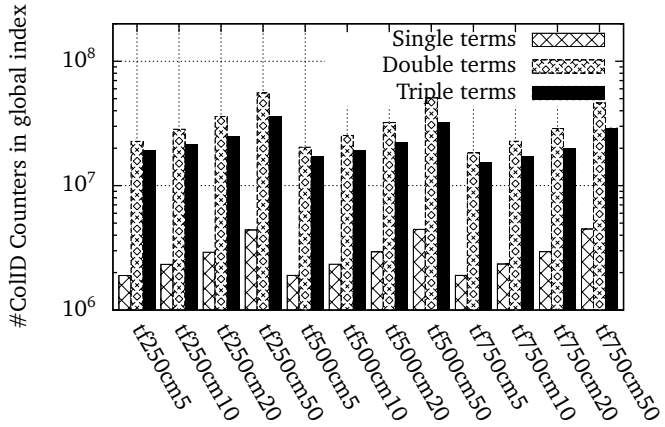

Fig. D.2. Number of collection ID counters with Random 100.

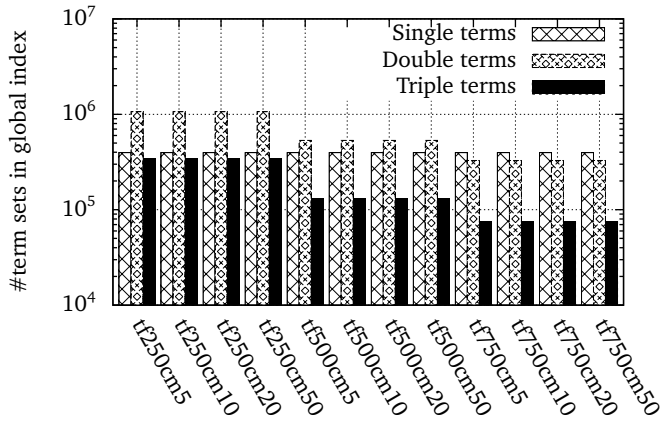

Fig. D.3. Number of term sets with Random 11512.

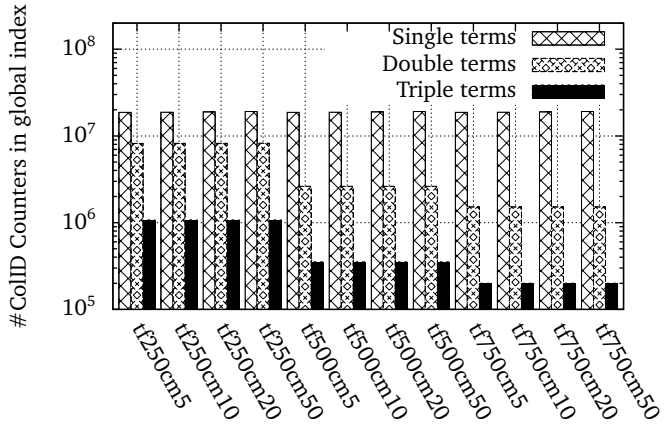

Fig. D.4. Number of collection ID counters with Random 11512.

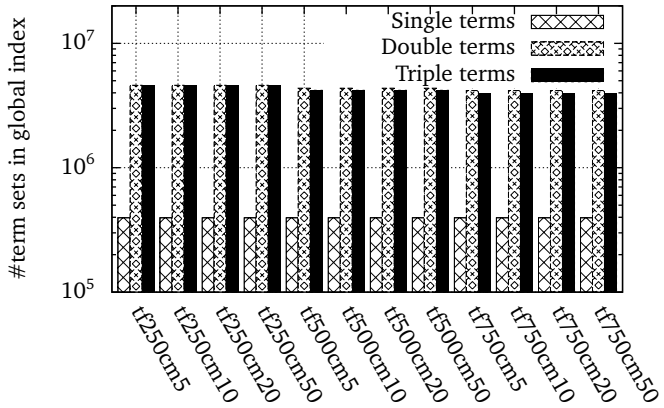

Fig. D.5. Number of term sets with IP Merge 100.

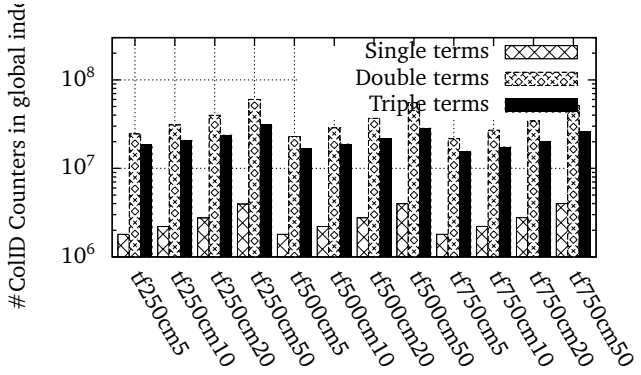

Fig. D.6. Number of collection ID counters with IP Merge 100.

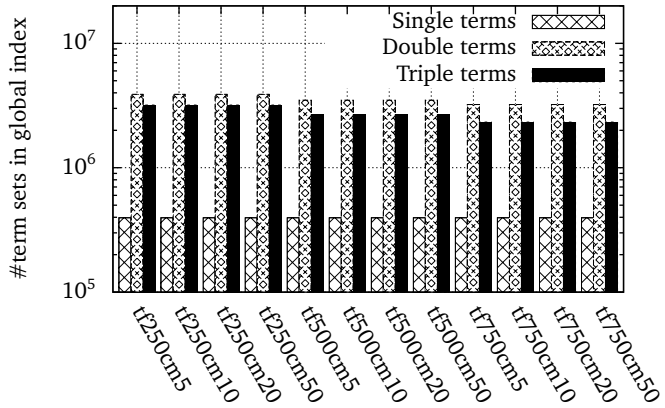

Fig. D.7. Number of term sets with IP Split.

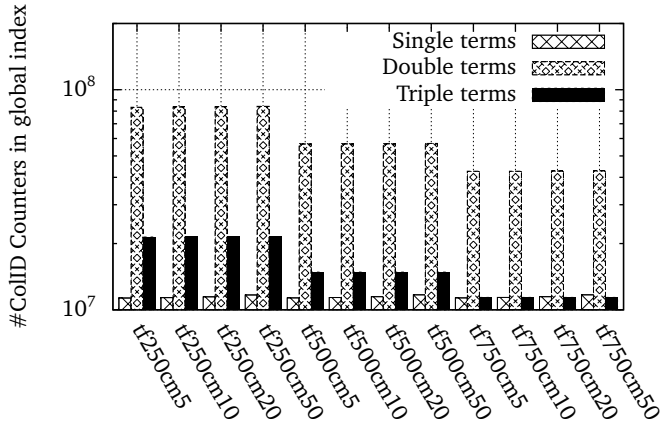

Fig. D.8. Number of collection ID counters with IP Split.

Nomenclature

cm	The collection maximum is a global indexing parameter. It denotes the maximum number of collections that are stored for a term set in the global index.
tf or $tf_{\max}$	The term set frequency maximum is a local indexing parameter. It denotes the maximum number of term set occurrences before a term set is called frequent.
ws	The window size denotes the number of words *after* a particular word with which combinations of term sets can be made with the first word.
LMDS	Language modeling with Dirichlet smoothing
ColID counter	The collection identifier counter is a global index component. A particular term set may occur in multiple collections. The collection identifier counter is a tuple that stores a collection identifier and a frequency counter for a term set.
HDK	Highly discriminative key
merit	The number of relevant documents within a collection. The merit is calculated per query.
QDI	Query-driven indexing

References

1. Lada A. Adamic and Bernardo A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.
2. Brian Amento, Loren Terveen, and Will Hill. Does "authority" mean quality? predicting expert quality ratings of web documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 296–303. ACM New York, NY, USA, 2000.
3. Jaime Arguello, Jonathan L. Elsas, Jamie Callan, and Jaime G. Carbonell. Document representation and query expansion models for blog recommendation. In *Proc. of the 2nd Intl. Conf. on Weblogs and Social Media (ICWSM)*, 2008.
4. Ricardo Baeza-Yates and Gonzalo Navarro. *Modeling Text Databases*, pages 1–25. Springer, 2004.
5. Peter Bailey, Nick Craswell, and David Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Inf. Process. Manage.*, 39(6):853–871, 2003.
6. Protima Banerjee and Hyoil Han. Drexel at trec 2007: Question answering. In *The Sixteenth Text REtrieval Conference (TREC 2007) Proceedings*, 2007.
7. Judit Bar-Ilan. Position paper: Access to query logs – an academic researcher's point of view. In *16th International World Wide Web Conference (WWW'07)*, 2007.
8. Fabiano C. Botelho, Daniel Galinkin, Jr. Wagner Meira, and Nivio Ziviani. Distributed perfect hashing for very large key sets. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
9. Jeremy T. Bradley, Douglas V. de Jager, William J. Knottenbelt, and Aleksandar Trifunovic. Hypergraph partitioning for faster parallel pagerank computation. In *EPEW/WS-FM*, pages 155–171, 2005.
10. Sergey Brin and Larry Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.
11. Andrei Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.
12. Jamie Callan. Distributed information retrieval. *Advances in Information Retrieval*, pages 127–150, 2000.
13. Jamie Callan, W. Bruce Croft, and Stephen M. Harding. The inquery retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems Applications*, pages 78–83, Valencia, Spain, 1992. Springer-Verlag.

14. Jamie Callan, Zhihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–28, New York, NY, USA, 1995. ACM.
15. David. F Carr. How google works. *Baseline Magazine*, 2006.
16. Kevyn Collins-Thompson and Jamie Callan. Query expansion using random walk models. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 704–711, New York, NY, USA, 2005. ACM.
17. Kevyn Collins-Thompson and Jamie Callan. Estimation and use of uncertainty in pseudo-relevance feedback. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 303–310, New York, NY, USA, 2007. ACM.
18. Kevyn Collins-Thompson, Paul Ogilvie, and Jamie Callan. Initial results with structured queries and laguage models on half a terabyte of text. text retrieval conference. In *In Proc. of 2005 Text REtrieval Conference. NIST Special Publication*, 2004.
19. Nick Craswell, Peter Bailey, and David Hawking. Is it fair to evaluate web systems using trec ad hoc methods. In *Workshop on Web Evaluation*. SIGIR'99, 1999.
20. Nick Craswell, Peter Bailey, and David Hawking. Server selection on the world wide web. In *DL '00: Proceedings of the fifth ACM conference on Digital libraries*, pages 37–46, New York, NY, USA, 2000. ACM.
21. Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 57–66, 2005.
22. Edleno S. de Moura, Célia F. dos Santos, Daniel R. Fernandes, Altigran S. Silva, Pavel Calado, and Mario A. Nascimento. Improving web search efficiency via a locality based static pruning method. *Proceedings of the 14th international conference on World Wide Web*, pages 235–244, 2005.
23. Fernando D. Diaz. Improving relevance feedback in language modeling with score regularization. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 807–808, New York, NY, USA, 2008. ACM.
24. Daryl D'Souza, James A. Thom, and Justin Zobel. Collection selection for managed distributed document databases. *Information Processing & Management*, 40(3):527–546, May 2004.
25. Daryl D'Souza, Justin Zobel, and James A. Thom. Is cori effective for collection selection? an exploration of parameters, queries, and data. In *Proceedings of the 9th Australasian Document Computing Symposium*, pages 41–46, December 2004.
26. James C. French, Allison L. Powell, Jamie Callan, Charles L. Viles, Travis Emmitt, Kevin J. Prey, and Yun Mou. Comparing the performance of database selection algorithms. *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 238–245, 1999.
27. Jeffrey L. Goldberg. Cdm: an approach to learning in text categorization. *Tools with Artificial Intelligence, 1995. Proceedings., Seventh International Conference on*, pages 258–265, November 1995.
28. Luis Gravano and Hector Garcia-Molina. Generalizing GlOSS to vector-space databases and broker hierarchies. In *International Conference on Very Large Databases, VLDB*, pages 78–89, 1995.
29. Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. Gloss: text-source discovery over the internet. *ACM Trans. Database Syst.*, 24(2):229–264, 1999.

30. Antonio Gulli and Alessio Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902–903, New York, NY, USA, 2005. ACM.
31. David Hawking and Paul Thomas. Server selection methods in hybrid portal search. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 75–82, New York, NY, USA, 2005. ACM.
32. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., Orlando, FL, USA, 1978.
33. Djoerd Hiemstra. Information retrieval models. In *Information Retrieval: Searching in the 21st Century*. Wiley, 2009.
34. Djoerd Hiemstra and Wessel Kraaij. Twenty-one at trec-7: Ad-hoc and cross-language track. In *In Proc. of Seventh Text REtrieval Conference (TREC-7*, pages 227–238, 1999.
35. David A. Hull. Stemming algorithms: A case study for detailed evaluation. *Journal of the American Society for Information Science*, 47:70–84, 1996.
36. Sepandar D. Kamvar, Taher H. Haveliwala, Christopher D. Manning, and Gene H. Golub. Exploiting the block structure of the web for computing pagerank. Stanford University Technical Report, 2003.
37. Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46:668–677, 1999.
38. Fabius Klemm, Anwitaman Datta, and Karl Aberer. A query-adaptive partial distributed hash table for peer-to-peer systems. *International Workshop on Peerto-Peer Computing & DataBases (P2P&DB 2004), Crete, Greece, March*, 2004.
39. Wessel Kraaij and Renée Pohlmann. Viewing stemming as recall enhancement. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 40–48, New York, NY, USA, 1996. ACM.
40. Tessa Lau and Eric Horvitz. Patterns of search: analyzing and modeling web query refinement. In *UM '99: Proceedings of the seventh international conference on User modeling*, pages 119–128, Secaucus, NJ, USA, 1999. Springer-Verlag New York, Inc.
41. Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107, 1999.
42. R. Lempel and S. Moran. The stochastic approach for link-structure analysis (salsa) and the tkc effect. *Computer Networks*, 33(1-6):387–401, 2000.
43. Jie Lu and Jamie Callan. Pruning long documents for distributed information retrieval. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 332–339, New York, NY, USA, 2002. ACM.
44. Toan Luu, Fabius Klemm, Martin Rajman, and Karl Aberer. Using Highly Discriminative Keys for Indexing in a Peer-to-Peer Full-Text Retrieval System. Technical report, Technical Report 2005041, School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne (EPFL), 2005, 2005.
45. Peter Lyman and Hal R. Varian. How much information, 2003. http://www.sims.berkeley.edu/how-much-info-2003, retrieved on April 23, 2008.
46. Donald Metzler. Indri retrieval model overview, July 2005. http://ciir.cs.umass.edu/~metzler/indriretmodel.html, retrieved on January 20, 2008.
47. Donald Metzler and W. Bruce Croft. Combining the language model and inference network approaches to retrieval. *Information Processing and Management*, 40(5):735–750, 2004.

48. Josiane Xavier Parreira and Gerhard Weikum. Jxp: Global authority scores in a p2p network. *International Workshop on Web and Databases, Baltimore, USA*, 2005.
49. Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 1, New York, NY, USA, 2006. ACM.
50. Ivana Podnar Zarko, Toan Luu, Martin Rajman, Fabius Klemm, and Karl Aberer. A peer-to-peer architecture for information retrieval across digital library collections. *European conference on research and advanced technology for digital libraries (ECDL 2006)*, pages 14–25, 2006.
51. Ivana Podnar Zarko, Martin Rajman, Toan Luu, Fabius Klemm, and Karl Aberer. Beyond term indexing: A p2p framework for web information retrieval. *Informatica*, 2(30):153–161, 2006.
52. Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281, New York, NY, USA, 1998. ACM.
53. Martin F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.
54. Allison L. Powell and James C. French. Comparing the performance of collection selection algorithms. *ACM Trans. Inf. Syst.*, 21(4):412–456, 2003.
55. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.
56. Stephen E. Robertson, Steve Walker, Susan Jones, Micheline M. Hancock-beaulieu, and Mike Gatford. Okapi at trec-3. In *TREC-3 Proceedings*, pages 109–126, 1995.
57. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
58. Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.
59. Karthikeyan Sankaralingam, Madhulika Yalamanchi, Simha Sethumadhavan, and James C. Browne. Pagerank computation and keyword search on distributed systems and p2p networks. *Journal of Grid Computing*, 1(3):291–307, 2003.
60. Nobuyoshi Sato, Minoru Udagawa, Minoru Uehara, Yoshifumi Sakai, and Hideki Mori. Query based site selection for distributed search engines. *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 556–561, 2003.
61. Jacques Savoy and Yves Rasolofo. Report on the TREC-10 experiment: Distributed collections and entrypage searching. In *Text REtrieval Conference*, 2001.
62. Douglas C. Schmidt. *More C++ gems*, chapter GPERF: a perfect hash function generator, pages 461–491. Cambridge University Press, New York, NY, USA, 2000.
63. Jangwon Seo and W. Bruce Croft. Umass at trec 2007 blog distillation task. NIST, 2007.
64. Pavel Serdyukov. Query routing in peer-to-peer web search. Master's thesis, Saarland University, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2005.
65. Milad Shokouhi, Justin Zobel, Saied Tahaghoghi, and Falk Scholer. Using query logs to establish vocabularies in distributed information retrieval. *Information Processing and Management*, 43(1):169–180, 2007.
66. Luo Si and Jamie Callan. Relevant document distribution estimation method for resource selection. In *SIGIR '03: Proceedings of the 26th annual international ACM*

SIGIR conference on Research and development in informaion retrieval, pages 298–305, New York, NY, USA, 2003. ACM.

67. Luo Si, Rong Jin, Jamie Callan, and Paul Ogilvie. A language modeling framework for resource selection and results merging. *Proceedings of the eleventh international conference on Information and knowledge management*, pages 391–397, 2002.
68. Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.
69. Gleb Skobeltsyn and Karl Aberer. Distributed cache table: efficient query-driven processing of multi-term queries in p2p networks. In *P2PIR '06: Proceedings of the international workshop on Information retrieval in peer-to-peer networks*, pages 33–40, New York, NY, USA, 2006. ACM.
70. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Zarko, Martin Rajman, and Karl Aberer. Query-Driven indexing for Peer-to-Peer Text Retrieval. In *16th International World Wide Web Conference (WWW'2007)*, 2007.
71. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Zarko, Martin Rajman, and Karl Aberer. Web text retrieval with a p2p query-driven index. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 679–686, New York, NY, USA, 2007. ACM.
72. Gleb Skobeltsyn, Toan Luu, Ivana Podnar Zarko, Martin Rajman, and Karl Aberer. Query-driven indexing for scalable peer-to-peer text retrieval. *Future Generation Computer Systems*, 25(1):89–99, June 2009.
73. Ian Soboroff. Does wt10g look like the web? In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 423–424, New York, NY, USA, 2002. ACM.
74. Karen Spärck Jones et al. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.
75. Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. Searching the web: the public and their queries. *J. Am. Soc. Inf. Sci. Technol.*, 52(3):226–234, 2001.
76. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.
77. Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligence Analysis*, 2004.
78. Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186, 2003.
79. Koen J. Tinselboer. The use of rare key indexing for distributed web search. Master's thesis, University of Twente, September 2007.
80. Panayiotis Tsaparas. *Link analysis ranking*. PhD thesis, Toronto, Ont., Canada, Canada, 2004. Adviser-Allan Borodin.
81. Yuan Wang and David J. DeWitt. Computing pagerank in a distributed internet search system. In Proceedings of the International Conference on Very Large Databases (VLDB), August 2004.

82. Jie Wu and Karl Aberer. Using siterank for p2p web retrieval. Technical Report IC/2004/31, Swiss Federal Institute of Technology, Lausanne, Switzerland, March 2004.
83. Jinxi Xu and W. Bruce Croft. Cluster-based language models for distributed retrieval. *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261, 1999.
84. Xing Yi and James Allan. Indri at trec 2007: Million query (1mq) track. In *The Sixteenth Text REtrieval Conference (TREC 2007) Proceedings*, 2007.
85. Budi Yuwono and Dik Lun Lee. Server ranking for distributed text retrieval systems on the internet. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 41–50. World Scientific Press, 1997.
86. ChengXiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 334–342, New York, NY, USA, 2001. ACM.
87. Yangbo Zhu, Shaozhi Ye, and Xing Li. Distributed pagerank computation based on iterative aggregation-disaggregation methods. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 578–585, New York, NY, USA, 2005. ACM.