Efficient Query Processing and Index Tuning using Proximity Scores

Dissertation
zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Andreas Broschart

Universität des Saarlandes

Saarbrücken
2012

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Mark Groves
Vorsitzender der Prüfungskommission	Prof. Dr.-Ing. Thorsten Herfet
Berichterstatter	PD Dr.-Ing. Ralf Schenkel
Berichterstatter	Prof. Dr.-Ing. Gerhard Weikum
Berichterstatter	Prof. Torsten Suel, PhD
Beisitzer	Dr.-Ing. Klaus Berberich
Tag des Promotionskolloquiums	09.10.2012

Acknowledgments

I would like to to express my sincere gratitude to my supervisor, PD Dr.-Ing. Ralf Schenkel for guiding me from my Master's to my PhD degree. He has always been a source of motivation and the door of his office has never been closed whenever I needed support. I would like to thank him for many interesting and fruitful discussions as well as for the scientific guidance he gave me.

My special thanks go to Prof. Dr.-Ing. Gerhard Weikum for giving me the opportunity to pursue my PhD studies in Saarbrücken, for his helpful comments, and for joining the reviewers board.

I would like to also thank Prof. Torsten Suel, PhD for accepting my request to review my PhD thesis.

Furthermore, I would like to thank Prof. Dr.-Ing. Thorsten Herfet for chairing the examination board and Dr.-Ing. Klaus Berberich for taking the minutes.

Finally, I would like to thank my colleagues for the great atmosphere at the work place and many enjoyable moments.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 31.05.2012

(Unterschrift)

Kurzfassung

Angesichts wachsender Datenmengen stellt effiziente Anfrageverarbeitung, die gleichzeitig Ergebnisqualität und Indexgröße berücksichtigt, zusehends eine Herausforderung für Suchmaschinen dar. Wir zeigen, wie man Proximityscores einsetzen kann, um Anfragen effektiv und effizient zu verarbeiten, wobei der Schwerpunkt auf eines der Ziele gelegt wird.

Die Hauptbeiträge dieser Arbeit gliedern sich wie folgt:

- Wir präsentieren eine umfassende vergleichende Analyse von Proximityscoremodellen sowie eine gründliche Analyse des Potenzials von Phrasen und passen ein führendes Proximityscoremodell für die Verwendung mit XML-Daten an.
- Wir diskutieren für die präsentierten Proximityscoremodelle die Eignung zur Top-k-Anfrageverarbeitung und präsentieren einen neuen Index, der einen Inhalts- und Proximityscore kombiniert, um Top-k-Anfrageverarbeitung zu beschleunigen und die Güte zu verbessern.
- Wir präsentieren ein neues, verteiltes Indextuningpaket für Term- und Termpaarlisten, das Tuningparameter mittels wohldefinierter Optimierungskriterien unter Größenbeschränkung bestimmt. Indizes können auf Effizienz oder Güte optimiert werden und sind bei hoher Güte performant.
- Wir zeigen, dass gekürzte Indizes mit einem Merge Join-Ansatz Top-k Algorithmen mit ungekürzten Indizes bei hoher Güte schlagen.
- Außerdem präsentieren wir eine hybride Indexstruktur, die Cold Cache-Effizienz verbessert.

Abstract

In the presence of growing data, the need for efficient query processing under result quality and index size control becomes more and more a challenge to search engines. We show how to use proximity scores to make query processing effective and efficient with focus on either of the optimization goals.

More precisely, we make the following contributions:

- We present a comprehensive comparative analysis of proximity score models and a rigorous analysis of the potential of phrases and adapt a leading proximity score model for XML data.
- We discuss the feasibility of all presented proximity score models for top-k query processing and present a novel index combining a content and proximity score that helps to accelerate top-k query processing and improves result quality.
- We present a novel, distributed index tuning framework for term and term pair index lists that optimizes pruning parameters by means of well-defined optimization criteria under disk space constraints. Indexes can be tuned with emphasis on efficiency or effectiveness: the resulting indexes yield fast processing at high result quality.
- We show that pruned index lists processed with a merge join outperform top-k query processing with unpruned lists at a high result quality.
- Moreover, we present a hybrid index structure for improved cold cache run times.

Zusammenfassung

Auf der Suche nach Information erwarten Leute qualitativ hochwertige Ergebnisse bei schnellen Antwortzeiten, zwei gegensätzliche Ziele. Angesichts ständig wachsender Datenmengen wird dabei effiziente Anfrageverarbeitung unter Berücksichtigung von Ergebnisgüte und Indexgröße zusehends eine Herausforderung für Suchmaschinen. Diese Arbeit beschäftigt sich mit dem wichtigen Problem, wie man Proximityscoremodelle einsetzen kann, um Anfrageverarbeitung gleichzeitig effizient und effektiv zu machen. Wir stellen neuartige Indexstrukturen vor, die Top-k-Anfrageverarbeitung erlauben und auf eine Reihe von Proximityscoremodellen anwendbar sind. Ein neuartiger Algorithmus zur Indexoptimierung kann für Ergebnisgüte oder Effizienz unter Indexgrößenkontrolle optimiert werden.

Der erste Teil dieser Arbeit widmet sich hauptsächlich Effektivitätsaspekten von Proximityscoremodellen.

In einer groß angelegten Studie existierender Proximityscoremodelle klassifizieren wir diese in vier Kategorien: 1) Linearkombinationen eines Inhaltsscoremodells und eines Proximityscoremodells, 2) integrierte Scoremodelle, 3) Sprachmodelle mit Wortabstandskomponenten und 4) Modelle, die Wortabstandsinformation verwenden und Scoremodelle mit maschinellen Lerntechniken erlernen. Wir präsentieren eine vergleichende Effektivitätsanalyse für eine beträchtliche Menge von Proximityscoremodellen, die wir in einem gemeinsamen Framework mit Hilfe von vier Testumgebungen evaluieren. Wir führen einen systemorientierten Vergleich der erforderlichen Features jedes Scoremodells durch. Für jedes Scoremodell in jeder Testumgebung empfehlen wir Modellparameter, die eine hohe Ergebnisgüte erzielen. Zusätzlich evaluieren wir, wie in [Met06a] vorgeschlagen, die Robustheit jedes Modells bezüglich Modellparametern.

Weiterhin führen wir einige Erweiterungen für Proximityscores in der Textsuche durch. Die Verwendung von Phrasen als hartes Filterkriterium für Ergebnisse ist eine weitere Möglichkeit, die Ergebnisqualität zu verbessern. Wir führen eine gründliche Analyse des Potenzials expliziter Phrasen für die Ergebnisqualität durch und vergleichen sie mit der Ergebnisqualität eines der führenden Proximityscoremodelle. Eine Nutzerstudie untersucht, wie sehr Nutzer bei der Kennzeichnung von Phrasen für eine gegebene Anfrage übereinstimmen. Wir validieren die weit verbreitete Intuition, dass die Verwendung von Phrasen in Anfragen die Ergebnisqualität existierender Retrievalmodelle steigern kann. Jedoch ist die Wahl geeigneter Phrasen eine nichttriviale Aufgabe und kann unter Umständen zu schwierig für Benutzer sein, die zudem häufig über die Wahl geeigneter Phrasen uneins sind. Weiterhin kommt der Anordnung von Termen in Anfragen nicht immer eine semantische Bedeutung zu.

Aufgrund der Verbreitung von XML-Dokumenten ist es nützlich, dass eine Suchmaschine nicht nur unstrukturierte Textdokumente unterstützt, sondern auch semistrukturierte XML-Daten. Wir passen eines der besten Proximityscoremodelle aus der Textsuche an, um Inhaltsanfragen auf XML-Daten zu unterstützen. Mit Hilfe zusätzlicher Abstände an den Elementgrenzen tragen wir der Dokumentstruktur für die XML-Elementsuche Rechnung, wenn wir die Distanz von Termauftreten berechnen.
Der zweite Teil dieser Arbeit konzentriert sich auf Effizienzaspekte von Proximityscoremodellen. Nachdem wir eine Einführung in Top-k und Nicht-Top-k Algorithmen gegeben haben, passen wir eines der führenden Proximityscoremodelle so an, dass wir es vorberechnen und in eine Indexstruktur gießen können. Weiterhin diskutieren wir für alle im ersten Teil vorgestellten Proximityscoremodelle die Anwendbarkeit der zuvor entwickelten Methoden. Wo diese anwendbar sind, leiten wir obere und untere Scoreschranken für Kandidaten und Zwischenergebnisse in Top-k Algorithmen her und entwickeln passende Indizes. Wir zeigen weiterhin, dass bereits wenige Tausend gelesene Indexeinträge hinreichend sind, um eine Ergebnisgüte zu erzielen, die mit ungekürzten Indizes erreicht werden kann. Weil derart wenige Einträge gelesen werden müssen, eröffnet das die Möglichkeit, auf gekürzten Indexlisten einen Merge Join-basierten Ansatz zu verwenden. Das spart gleichzeitig zusätzliche Kosten der Top-k-Algorithmen und reduziert signifikant die Indexgröße. Wir erzielen beeindruckende Effizienzsteigerungen um bis zu zwei Größenordnungen verglichen mit dem Lesen ungekürzter Listen mit Inhaltsscoreinformation. In einem ersten Ansatz wurden dabei alle Listen auf nichtsystematische Art gekürzt.

Das ist die Stelle, an der unser Indexoptimierungspaket für Term- und Termpaarindexlisten ins Spiel kommt. Wir schlagen einen systematischen Ansatz vor, der mit wohldefinierten Optimierungskriterien Parameter zum Kürzen von Indexlisten errechnet. Dazu entwickeln wir ein Indexoptimierungspaket, das Indexstrukturen für Terme und Termpaare für maximale Ergebnisgüte oder maximale Effizienz unter Gütekontrolle und Indexgrößenbudget optimiert. Das Paket verwendet Hadoop, ein Open Source MapReduce-Paket und gestattet eine selektive Materialisierung von Termpaarlisten auf der Basis von Information aus einer Anfrageprotokolldatei. Wir zeigen, wie wir Indizes sowohl mit als auch alternativ ohne Bewertungen der Ergebnisrelevanz optimieren können. Die resultierenden gekürzten Indizes bieten verlässliche Anfrageausführungszeiten und eine Ergebnisgüte, die vergleichbar oder sogar besser als die ungekürzter Termindizes ist, welche die Ergebnisgüte des BM25-Bewertungsmodells liefern. Wir präsentieren eine hybride Indexstruktur, welche Term- und Termpaarindexlisten kombiniert, um weniger gelesene Listen gegen eine höhere gelesene Datenmenge einzutauschen, um Cold Cache-Laufzeiten zu verbessern. Wir zeigen experimentell, dass die resultierenden gekürzten Indizes Anfragen um fast eine Größenordnung gegenüber einem führenden Top-k Algorithmus bei vergleichbarer Ergebnisgüte beschleunigen. Wir führen ausgedehnte Experimente auf den Dokumentkollektionen GOV2 und ClueWeb09 sowie für den INEX Efficiency Track 2009 und den TREC Web Track 2010 durch.

Summary

When people search for information, they expect high quality results at fast processing times which are conflicting goals. In the presence of growing data, the need for efficient query processing under result quality and index size control increasingly becomes a challenge to search engines. This work addresses the important problem how to use proximity scores to make query processing effective and efficient at the same time. We present novel index structures for top-k query processing applicable to a number of proximity score models and a novel algorithm for index tuning that can be optimized for retrieval quality or efficiency under index size control.

The first part of this thesis deals mainly with effectiveness aspects of proximity score models.

In an extensive survey of existing proximity-enhanced score models, we put them into four categories: 1) linear combinations of a content score model and a proximity score model, 2) integrated score models, 3) language models with proximity components, and 4) models that incorporate proximity features and learn to rank by application of machine learning techniques. We present a comparative analysis of a significant set of proximity score models in a single evaluation framework with four test beds. We carry out a system-oriented comparison with the required features per score model. We give recommendations on how to set parameters for each combination of test bed and score model. In addition, we measure intercollection and intracollection generalization, entropy, and spread values as proposed in [Met06a].

Furthermore, we elaborate on some extensions to proximity scores in text retrieval. Usage of phrases as a hard filter criterion for results is a different means to improve retrieval quality. We carry out a rigorous analysis of the potential of explicit phrases for retrieval quality and compare it to the retrieval quality of a state-of-the-art proximity score model. A user study investigates the degree of user agreement about phrases in a query. We validate the common intuition that phrase queries can boost the performance of existing retrieval models, but choosing good phrases is a non-trivial task and might be too difficult for users as they frequently disagree on phrases in a query; furthermore, term order in queries does not always bear semantics.

Due to the dissemination of XML documents, it is useful for a search engine to not only support unstructured text documents, but also semi-structured XML data. We adapt one of the best performing proximity score models from text retrieval to support content queries on XML data. By means of virtual gaps in XML documents, we take

the document structure into account when computing the distance of term occurrences.

The second part of this thesis concentrates on efficiency aspects of proximity score models.

After giving an introduction into top-k and non-top-k algorithms, we show how to adapt a state-of-the-art proximity score model for top-k query processing and devise appropriate index structures that allow precomputation of the required features. Furthermore, we discuss the feasibility of all proximity score models presented in the survey for top-k query processing, give score bounds, and devise indexes where possible. We furthermore show that already a few thousand read entries are good enough to yield a retrieval quality comparable to reading unpruned index lists. As only that few entries have to be read, this opens the door to merge join processing on pruned index lists, saving on overhead costs of top-k query processing and index space requirements. We achieve impressive performance gains by up to two orders of magnitude compared to reading unpruned content score lists. However, all index lists have been pruned in a non-systematic, ad hoc style manner.

That is the place where our index tuning framework for term and term pair index lists comes into play. We propose a systematic pruning approach with well-defined optimization criteria. To this end, we introduce a tunable indexing framework for term and term pair index structures for optimizing index parameters towards either maximal result quality or maximal query processing performance under result quality control, given a maximal index size. The index tuning framework is implemented on top of the Open Source MapReduce framework Hadoop and allows a selective materialization of term pair index lists based on information from a query log. We show how to perform index tuning both in the presence and, alternatively, in the absence of relevance assessments. The resulting indexes provide dependable query execution times while providing result quality comparable to or even better than unpruned term indexes that provide BM25 score quality. We present a hybrid index structure that combines the term and term pair index lists to trade in a reduced number of fetched lists for an increased number of read bytes to improve cold cache run times. Experimental results demonstrate that the resulting index configurations allow query processing that achieves almost one order of magnitude performance gain compared to a state-of-the-art top-k algorithm yielding results of comparable quality. We carry out extensive experiments on GOV2 and ClueWeb09, in the INEX 2009 Efficiency Track and for the TREC Web Track 2010.

Contents

List of Abbreviations

$\mathcal{C} = \{d_1, \ldots, d_N\}$: *document corpus/collection consisting of* N *documents*

$ctf(t_i) = \sum_{d \in \mathcal{C}} tf(t_i, d)$: *collection term frequency of term* t_i *in* $\mathcal{C}$

$df(t_i)$: *document frequency of term* t_i *in* $\mathcal{C}$

dt : *number of distinct terms in* $\mathcal{C}$

$dt(d)$: *number of distinct terms in document* d

$idf(t_i)$: *inverse document frequency of term* t_i *in* $\mathcal{C}$

$idf_j(t_i)$: *inverse document frequency of term* t_i *in* $\mathcal{C}$*, variant* j

$l_{\mathcal{C}} = \sum_{d \in \mathcal{C}} l_d$: *length of document collection* $\mathcal{C}$

$l_d = |d|$: *length of document* d

$l_e = |e|$: *length of element* e

$K = k \cdot [(1-b) + b \cdot \frac{l_d}{avgdl}]$: *frequently occurring component in scoring models*

N : *number of documents in the document corpus/collection* $\mathcal{C}$

$P \subseteq \{1, \ldots, l_d\}$: *subset of positions in document* d

$P_d(t) = \{i : p_i(d) = t\} \subseteq \{1, \ldots, l_d\}$: *set of positions in document* d *where term* t *occurs*

$P_e(t) = \{i : p_i(e) = t\} \subseteq \{1, \ldots, l_e\}$: *set of positions in element* e *where term* t *occurs*

$P_d(q) := \cup_{t_i \in q} P_d(t_i)$: *set of positions of all query terms in document* d

$P_e(q) := \cup_{t_i \in q} P_e(t_i)$: *set of positions of all query terms in element* e

$p_i(d_j)$: *term occurring at position* i *of document* d_j

$p_i(e_j)$: *term occurring at position* i *of element* e_j

$q = \{t_1, \ldots, t_n\}$: *unordered query with query terms* $t_1, \ldots, t_n$

$Q_{adj,d}(q) := \{(i,j) \in P_d(q) \times P_d(q) \mid (i < j) \wedge \forall k \in \{i+1, \ldots, j-1\} : k \notin P_d(q)\}$:
set of pairs of query terms in document d *that are adjacent to each other*

$Q_{adj,e}(q) := \{(i,j) \in P_e(q) \times P_e(q) \mid (i < j) \wedge \forall k \in \{i+1, \ldots, j-1\} : k \notin P_e(q)\}$:
set of pairs of query terms in element e *that are adjacent to each other*

$Q_{all,d}(q, dist) := \{(i,j) \in P_d(q) \times P_d(q) \mid (i < j) \wedge (j - i \leq dist)\}$:
set of pairs of query terms in document d *within a window of* $dist$ *positions*

$Q_{all,d}(q)$: *the same as* $Q_{all,d}(q, dist)$*, but employs a window size of* $dist = l_d$

$Q_{all,e}(q, dist) := \{(i,j) \in P_e(q) \times P_e(q) \mid (i < j) \wedge (j - i \leq dist)\}$:
set of pairs of query terms in element e *within a window of* $dist$ *positions*

$Q_{all,e}(q)$: *the same as* $Q_{all,e}(q, dist)$*, but employs a window size of* $dist = l_e$

$qtf(t_i)$: *query term frequency of term* t_i *in a query*

$S_q = (t_1, \ldots, t_n)$: *ordered query*

$T_d(P) = \{t|\ i \in P \wedge p_i(d) = t\}$: *terms located at the positions of P in document d*

$T_e(P) = \{t|\ i \in P \wedge p_i(e) = t\}$: *terms located at the positions of P in element e*

$tf(t_i, d)$: *term frequency of t_i in d*

$V = \{v_1, \ldots, v_m\}$: *vocabulary, set of terms which occur in an index*

$W_q = (qw(t_1), \ldots, qw(t_n)) \subset [0,1]^n$: *query term weights for terms in query q*

Chapter 1

Introduction

There is a plethora of applications, on the Web, in XML retrieval, in Intranets, Digital Libraries, or Desktop search, where large document collections need to be queried. Users expect not only high quality answers but also require almost instant response times. To achieve these conflicting goals, index structures and algorithms have to be devised that index documents in a compact way that allows determining a ranking of the top matching documents without inspecting the entire index. In this thesis, we focus on retrieval models for proximity search, which go far beyond simple bag of words. Proximity score models are a means to improve the retrieval quality of results by exploiting term position information of query term occurrences in documents where positional distances consider contextual information. Clearly, a good proximity score model has also to be robust to model parameters. Phrases are a hard filter for documents that can be used to further improve retrieval quality, but may also be subject to deleting potentially relevant results if the phrase in the query is not exactly matched in a document. Proximity scores allow soft phrase querying without the requirement to specify phrases.

The improvement in user-perceived result quality comes, however, in general at the price of a larger index size and higher query response times. As there is no need to exhaustively compute the score of all documents with respect to a query, as only the top ranked documents are shown to a user, we apply top-k algorithms which are an effective means to tackle efficiency issues by dynamic pruning/early termination. The key idea is to stop the query processing at a point where all potential top results have been inspected. In this context, it is important that a proximity score model can be cast into precomputed index lists to compute score bounds for result candidates, hence, allow early stopping. Devising compact index structures that can be efficiently queried and at the same time provide highly accurate results is the task we consider in this thesis. We show how proximity scores that enhance retrieval quality can be integrated into efficient top-k algorithms.

We propose to extend the index with additional term pair lists that maintain proximity scores. However, an index with these lists can become prohibitively large.

A naive approach would simply cut index lists or exclude complete lists already during the indexing phase. However, it remains unclear where to cut index lists, hence the tradeoff between performance gains and loss in user-perceived result quality is rather ad-hoc and bears the risk to drastically favor one or the other extreme.

To overcome this, in this thesis, we devise a number of techniques for limiting the index size. Occurrences within a large proximity distance have only a marginal contribution to the overall score, we propose a window-based pruning approach that only considers term pair occurrences in a text-window of fixed size. We heuristically limit the list length to a constant number of entries, usually in the order of a few thousand entries. Further list pruning with quality guarantees is applied.

We show that pruned term and term pair lists provide a retrieval quality comparable to unpruned term lists. At the same time, this not only saves on disk space, but significantly accelerates query processing. We propose an index tuning framework that prunes term and term pair lists in a systematic fashion and we prune both list types by list length, term pair lists are additionally restricted to entries above a minimal proximity score contribution. If the disk space is limited, control over the space consumption of index structures is necessary. It is desirable to opt between index optimization towards maximum efficiency and maximum effectiveness given an index size constraint. Our approach allows tuning pruning parameters by using a set of queries and their relevance assessments for the collection to be indexed or, alternatively, if relevance assessments are not available, by a result overlap approach. In addition, query logs can be used to select term pair lists to be materialized. Using lossless index compression, the index size can be further decreased.

Although this thesis focuses on Web Retrieval scenarios for the evaluation of the presented approaches, the developed techniques are not only applicable to Web Retrieval, but also to other domains such as book search over digital libraries or Intranet search for enterprises that keep track of various kinds of documents such as blueprints and patents. In fact, we make a proximity score feasible for XML element retrieval and show that we can apply our index tuning framework for indexes that support content queries for XML element retrieval. Beyond the technical contributions in the area of proximity indexing and search, this thesis provides a comprehensive survey that describes and experimentally compares a significant portion of proximity scoring models.

1.1 Contributions

1. We present a comprehensive comparative analysis of a significant set of proximity score models in a single evaluation framework with four test beds. We extensively present and classify existing proximity-enhanced score models in a joint notation; using one running example, we illustrate the various models and include a feature list to compare the required model features. We show how to adapt a state-of-the-art proximity scoring model to support content queries on XML data.

2. We carry out a rigorous analysis of the potential of explicit phrases for retrieval quality and compare it to the retrieval quality of a state-of-the art proximity score

model. A user study investigates the degree of user agreement about phrases in a query.

3. We propose a novel index structure that combines content and proximity scores. Processing that index structure together with a content score index improves query processing in top-k algorithms by up to two orders of magnitude through tighter score bounds and a better retrieval quality compared to processing content score lists only. We apply top-k query processing to several proximity score models and devise appropriate index structures.

4. We show that already a few thousand read entries on unpruned term and term pair lists are good enough to yield a retrieval quality comparable to reading unpruned index lists. This insight opens the door to a simple merge join-based approach with pruned index lists: we require less disk space and keep the performance improvements.

5. We propose a novel, distributed index tuning framework for term and term pair index lists that optimizes pruning parameters for retrieval quality or efficiency under index size control with well-defined optimization criteria. We allow a selective materialization of term pair index lists based on information from a query log and show how to perform index tuning both in the presence and in the absence of relevance assessments.

6. We present a hybrid index structure for improved cold cache run times of small and medium-sized queries that reduces the number of fetched index lists.

1.2 Publications

Various aspects of this thesis have been published in [SBH$^+$07, BS08b, BS08a, BST08, BS09, BBS10, BS10, BS11, BS12].

Effectiveness-related contributions have been described in the following publications: in [BS08b], we have presented a proximity score model for content-only queries on XML data, enriched with additional experiments on a different test bed in [BST08].

[BS08b] Andreas Broschart and Ralf Schenkel. Proximity-Aware Scoring for XML Retrieval. In Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Poster.

[BST08] Andreas Broschart, Ralf Schenkel, and Martin Theobald. Experiments with Proximity-Aware Scoring for XML Retrieval at INEX 2008. In Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008.

In [BBS10], we have rigorously analyzed the potential of phrases, compared the retrieval quality to proximity scores and carried out a user study.

[BBS10] Andreas Broschart, Klaus Berberich, and Ralf Schenkel. Evaluating the Potential of Explicit Phrases for Retrieval Quality. In Advances in Information Retrieval, 32nd European Conference on IR Research, ECIR 2010, Poster.

Efficiency-related contributions have been described in the following publications: in [SBH+07], we have shown how to accelerate top-k query processing by means of a content score index structure and a new index structure that incorporates a content and a proximity score. Furthermore, we have shown that a few thousand entries per list are enough to provide the same retrieval quality as on unpruned content score lists. In addition, pruning saves on index space.

[SBH+07] Ralf Schenkel, Andreas Broschart, Seung-Won Hwang, Martin Theobald, and Gerhard Weikum. Efficient Text Proximity Search. In String Processing and Information Retrieval, 14th International Symposium, SPIRE 2007.

In [BS08a], we have presented the merge join-based approach with pruned index lists to save on overhead costs of top-k query processing and to lower index space requirements.

[BS08a] Andreas Broschart and Ralf Schenkel. Effiziente Textsuche mit Positionsinformation. In Grundlagen von Datenbanken, 2008.

In [BS12], we have presented a novel, distributed index tuning framework which is a major part of this thesis (cf. Chapter 8) and supported it with extensive experiments especially for GOV2. Additional experiments with this tuning approach for more test beds have been released for the INEX 2009 Efficiency Track in [BS09] and for the TREC Web Track 2010 in [BS10].

[BS12] Andreas Broschart and Ralf Schenkel. High-Performance Processing of Text Queries with Tunable Pruned Term and Term Pair Indexes. In ACM Transactions on Information Systems 2012, Volume 30, Issue 1.

[BS09] Andreas Broschart and Ralf Schenkel. Index Tuning for Efficient Proximity-Enhanced Query Processing. In Focused Retrieval and Evaluation, 8th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2009.

[BS10] Andreas Broschart and Ralf Schenkel. MMCI at the TREC 2010 Web Track. In The Nineteenth Text REtrieval Conference Proceedings, TREC 2010.

In [BS11], we have presented a hybrid index structure for improved cold cache run times for pruned indexes from our index tuning framework that trades in a reduced number of fetched lists for an increased number of read bytes.

[BS11] Andreas Broschart and Ralf Schenkel. A Novel Hybrid Index Structure for Efficient Text Retrieval. In Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Poster.

1.3 Thesis Outline

Chapter 2 gives an extensive overview over proximity-enhanced score models that we put in four categories: after describing unigram models that serve as basis for proximity score models, we detail every proximity score model and carry out a system-oriented comparison of the implementation effort required per score model. Chapter 3 introduces two popular evaluation initiatives, namely TREC (text retrieval), and INEX (XML retrieval) and two less popular, niche initiatives. We present a choice of test beds for each of them and performance metrics for both text/document retrieval and for XML retrieval. Chapter 4 shows experimental results of the score models from the original papers surveyed in Chapter 2. As they usually compare only a few of the score models, we perform a comparative analysis of a significant portion of proximity score models in a single evaluation framework using four test beds. Chapter 5 proposes one of the first XML score models that uses proximity information, rigorously analyzes the potential of explicit phrases for retrieval quality, and compares it to a proximity score. Chapter 6 presents various top-k algorithms from both the database systems and the IR community as well as non-top-k algorithms. Chapter 7 describes a modification of Büttcher et al.'s score model that allows to use it in a top-k style with dynamic pruning techniques that not only improves retrieval effectiveness but also efficiency compared to standard top-k algorithms. We show that already a few thousand read entries per index list yield a good retrieval quality. This opens the door to using light-weight n-ary merge joins to save on processing overhead. Moreover, we discuss the feasibility of the remaining proximity score models surveyed in Chapter 2 for top-k query processing and propose appropriate index structures where possible. Chapter 8 introduces our index tuning framework for trading off index size and result quality given an index size constraint. Chapter 9 concludes this thesis and outlines possible future research directions.

Chapter 2

Proximity Score Models

2.1 Introduction

2.1.1 Motivation

In search engines scoring functions play an important role to rank results supposed to answer user queries. Therefore, the quality of the scoring function is decisive to user satisfaction and success of the search engine. Nowadays, many search engines rely on some form of BM25 [RW94, RWHB^{+}95], a state-of-the-art content-based scoring model commonly used in probabilistic information retrieval. It incorporates tf values (*term frequency*, i.e., the number of a term's occurrences in a document) and *idf* values (*inverse document frequency*, i.e., the inverse of the number of documents that contain a term) plus document length information. Content-based scoring models usually represent documents using "bags of words" that consider all query term occurrences, but ignore the information where these query terms occur. That way those models ignoring positional information are foregoing the chances to leverage the power of *term proximity* information, i.e., measuring the distance between query term occurrences in a document and aggregating them into a proximity score to rank the document appropriately. If this valuable information is ignored, users might face unsatisfactory results. Suppose a user poses the query *surface area of a triangular pyramid.* Scoring functions that completely ignore proximity information may consider documents relevant that contain query terms frequently, but in different paragraphs that are likely to treat different topics: in a document related to geometric objects like the one depicted in Figure 2.1, the first paragraph might elaborate on the "volume of a triangular prism", while the second talks about the "volume of a square pyramid", and the third about the "surface area of a cylinder". Each of the query terms will individually occur quite frequently, but not in the user-intended context. From a user's point of view, formulating her information need as a phrase query might be the solution to prevent such results. As phrase queries are usually used as hard filters, documents that do not contain the phrase terms in the exact order (as they might be interleaved by a different term or appear in a slightly different order) are ignored. Unfortunately this comes at the expense of many discarded good results - documents carrying information about

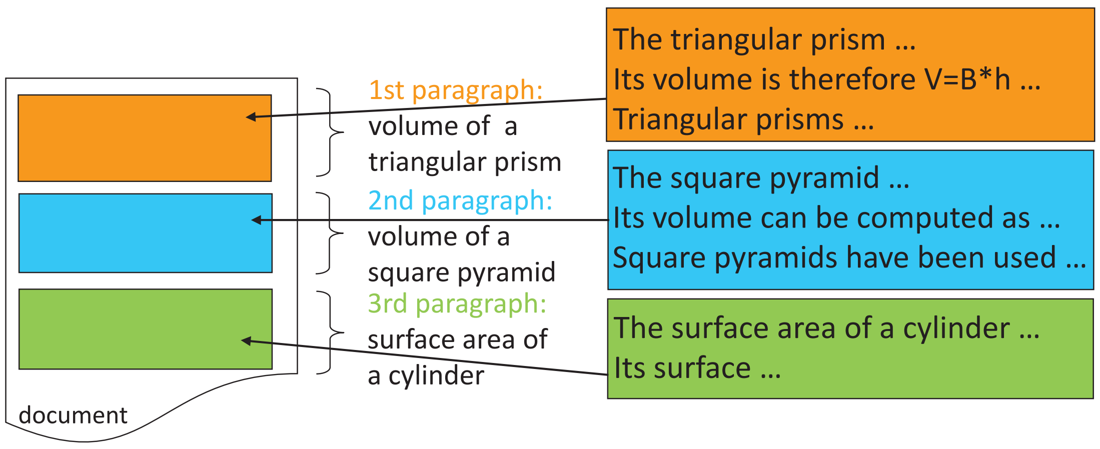

The triangular prism ...
Its volume is therefore V=B*h ...
Triangular prisms ...
1st paragraph:
volume of a triangular prism
The square pyramid ...
Its volume can be computed as ...
Square pyramids have been used ...
2nd paragraph:
volume of a square pyramid
The surface area of a cylinder ...
Its surface ...
3rd paragraph:
surface area of a cylinder
document

Figure 2.1: Non-relevant document for query *surface area of a triangular pyramid*.

the ”surface area of a pyramid composed of four triangular faces” would certainly be a good hit, but excluded by the phrase query. Proximity scores provide a solution to alleviate those effects by providing some kind of soft phrasing without the need to specify phrase bounds by the user.

This chapter gives an extensive overview of existing proximity-enhanced score models. We categorize them into the following four categories:

- linear combinations of a content score model and a proximity score model described in Section 2.4 (e.g., Rasolofo and Savoy [RS03], Büttcher et al. [BC05, BCL06], Uematsu et al. [UIF$^+$08], Monz [Mon04], and Tao and Zhai [TZ07]),

- integrated score models described in Section 2.5 (e.g., Song et al. [STW$^+$08], de Kretser and Moffat [dKM99, dKM04], and Mishne and de Rijke [MdR05]),

- language models with proximity components described in Section 2.6 (e.g., Lv and Zhai [LZ09], and Zhao and Yun [ZY09]), and

- models that incorporate proximity features and learn to rank by application of machine learning techniques described in Section 2.7 (e.g., Svore et al. [SKK10], Metzler and Croft [MC05], and Cummins and O’Riordan [CO09]).

An experimental study in Chapter 4 will investigate the retrieval quality for a selection of these approaches and compare them to the retrieval quality that can be achieved using BM25.

2.1.2 Model and Notation

In order to describe the scoring models of this survey in a uniform manner, we first introduce some notation.

Definition 2.1.1. (**corpus, vocabulary, position-related notation**)
A *corpus* $\mathcal{C} = \{d_1, \ldots, d_N\}$ is a set of N documents where each document is considered a sequence of terms. The *vocabulary* $V = \{v_1, \ldots, v_m\}$ is the set of terms which occur in an index. Given a document d with length l_d, we denote the *term occurring at position i of d* by $p_i(d), 1 \leq i \leq l_d$; if the document is clear from the context, we simply write p_i.
For a term t, we capture the *positions in document d where t occurs* by $P_d(t) = \{i : p_i(d) = t\} \subseteq \{1, \ldots, l_d\}$; if d is clear from the context, we write $P(t)$. We write $P_d(q) := \cup_{t_i \in q} P_d(t_i)$ for the *positions of all query terms in document d*, again omitting the suffix d if the document is clear from the context.
Given a set of positions $P \subseteq \{1, \ldots, l_d\}$ and a document d, we write $T_d(P)$ to denote the *set of terms at the positions of $P \subseteq \{1, \ldots, l_d\}$ in d.* Precisely, $T_d(P) := \{p_i(d) | i \in P\}$.

Definition 2.1.2. (**document frequency, inverse document frequency**)
A term t_i occurs in $df(t_i)$ documents in $\mathcal{C}$, the *document frequency of t_i*. The *inverse document frequency $idf(t_i)$* measures a term's importance in $\mathcal{C}$ by means of an inverse function of $df(t_i)$. In the literature (e.g., [BCL06, RS03, UIF+08]), the inverse document frequency $idf(t_i)$ is used in (slightly) different versions[1], e.g.,

- $idf_1(t_i) = \log \frac{N}{df(t_i)}$
- $idf_2(t_i) = max\{0, \log \frac{N - df(t_i)}{df(t_i)}\}$, and
- $idf_3(t_i) = max\{0, \log \frac{N - df(t_i) + 0.5}{df(t_i) + 0.5}\}$.

Definition 2.1.3. (**term frequency, collection term frequency for terms and n-grams**)
Given a term t_i, a corpus $\mathcal{C}$, and a document d in $\mathcal{C}$, the *term frequency of t_i in d*, $tf(t_i, d)$, is the number of times term t_i occurs in d. The *term frequency of the n-gram $(t_i, \ldots, t_{i+n-1})$ in d*, $tf((t_i, \ldots, t_{i+n-1}), d)$, is the number of times the n-gram $(t_i, \ldots, t_{i+n-1})$ occurs in d. The *collection term frequency of t_i* is the total number of occurrences of the term t_i in $\mathcal{C}$ and defined as $ctf(t_i) = \sum_{d \in \mathcal{C}} tf(t_i, d)$. The *collection term frequency of the n-gram $(t_i, \ldots, t_{i+n-1})$* is the total number of occurrences of the n-gram $(t_i, \ldots, t_{i+n-1})$ in $\mathcal{C}$ and defined as $ctf((t_i, \ldots, t_{i+n-1})) = \sum_{d \in \mathcal{C}} tf((t_i, \ldots, t_{i+n-1}), d)$.

Definition 2.1.4. (**document and collection length, number of distinct terms in a document or collection**)
Given a corpus $\mathcal{C}$ and a document d in $\mathcal{C}$, the *document length of d* corresponds to the number of term occurrences in d and is denoted by $l_d = |d|$. The *collection length* corresponds to the number of term occurrences in $\mathcal{C}$ and is denoted by $l_{\mathcal{C}} = \sum_{d \in \mathcal{C}} l_d$.
While $dt(d) = |\{t : tf(t, d) > 0\}|$ stands for the *number of distinct terms in document d*, we use $dt = |\{t : \exists d \in \mathcal{C} \text{ s.t. } tf(t, d) > 0\}|$ as an abbreviation for the *number of distinct terms in $\mathcal{C}$*.

[1]Please note that, unlike e.g., [RS03, UIF+08], for $idf_2(t_i)$ and $idf_3(t_i)$, we have imposed a lower bound of zero to avoid negative score contributions of too frequent terms.

For the ease of presentation, as a default, we assume that each query term occurs just once per query such that we can use sets of terms to model issued queries. In some cases it may be necessary to deviate from this (e.g., if the order of query terms in the original query matters). Where applicable we will make additional remarks in the description of the affected scoring model.

Definition 2.1.5. (**query, query term frequency, unordered query, ordered query, query term weights**)
W.l.o.g. the user issues an *(unordered) query* $q' = \{t'_1, \ldots, t'_x\}$ which is supposed to represent her information need; the query processor evaluates only query terms from V, i.e., the *evaluated query* is $q = q' \cap V = \{t_1, \ldots, t_n\}$. *Ordered queries* are denoted by $S_q = (t_1, \ldots, t_n)$.
The *query term frequency* denotes the number of times a query term t appears in a query S_q, short $qtf(t)$; for unordered queries $qtf(t)$ is either 0 or 1.
Query terms may be attributed *query term weights* $W_q = (qw(t_1), \ldots, qw(t_n)) \subset [0,1]^n$.

Definition 2.1.6. (**set of pairs of adjacent query term occurrences, set of pairs of all query term occurrences**)
We denote *pairs of query terms that are adjacent to each other* (there might be non-query terms in between) in document d by

$$Q_{adj,d}(q) := \{(i,j) \in P_d(q) \times P_d(q) \mid (i < j) \wedge \forall k \in \{i+1, \ldots, j-1\} : k \notin P_d(q)\}.$$

Pairs of query terms within a window of dist positions in document d are defined as

$$Q_{all,d}(q, dist) := \{(i,j) \in P_d(q) \times P_d(q) \mid (i < j) \wedge (j - i \leq dist)\}.$$

Please note that in this case, the query terms need not occur consecutively in a document. $Q_{all,d}(q) := Q_{all,d}(q, l_d)$ denotes all query term pairs in the document.

2.2 Unigram Models

This section describes unigram models that serve as basis for proximity scores.

2.2.1 BM25

We start with the probabilistic, content-scoring model BM25 [RW94, RWHB+95]. Robertson and Walker [RW94] define the relevance score of a document d for the query $q = \{t_1, \ldots, t_n\}$ as

$$score_{\text{BM25}}(d,q) = \sum_{t_i \in q} \frac{(k_1+1) \cdot tf(t_i,d)}{k \cdot [(1-b) + b \cdot \frac{l_d}{avgdl}] + tf(t_i,d)} \cdot W^{RSJ}(t_i) \cdot \frac{(k_3+1)qtf(t_i)}{k_3 + qtf(t_i)},$$

where the Robertson/Sparck Jones weight [RJ76] is defined as

$$W^{RSJ}(t_i) = \log \frac{(r(t_i)+0.5)(N - R - df(t_i) + r(t_i) + 0.5)}{(df(t_i) - r(t_i) + 0.5)(R - r(t_i) + 0.5)};$$

R denotes the number of relevant documents, and $r(t_i)$ the number of relevant documents which contain term t_i.
Later publications substitute the Robertson/Sparck Jones weight by a form of *idf* s.t.

$$score_{\text{BM25}}(d,q) = \sum_{t_i \in q} \frac{(k_1+1) \cdot tf(t_i,d)}{k \cdot [(1-b) + b \cdot \frac{l_d}{avgdl}] + tf(t_i,d)} \cdot idf(t_i) \cdot qtf'(t_i),$$

where *idf* and qtf' determine the specific variant of BM25; *idf* is a variant of the inverse document frequency (as described in Definition 2.1.2) and qtf' (cf. Definition 2.1.5) represents a function that incorporates t_i's query term frequency qtf. In the BM25 scoring model, k, k_1, and b are constants (where $k_1 = k$ in the original definition), and $avgdl$ is the average document length in the collection, i.e., $avgdl = \frac{l_C}{N}$. A frequently used abbreviation is $K = k \cdot [(1-b) + b \cdot \frac{l_d}{avgdl}]$.

Table 2.1 shows the *idf* and qtf' components and the tuning parameters k, k_1, b, and optionally k_3 as used in a follow-up [RWHB+95] to the original BM25 paper[2] and in some proximity scores. k_3 set to none indicates the absence of k_3 in the respective method. Additionally, the table contains a pointer to the section where the corresponding content/proximity scoring model is described.

method	*idf* or W^{RSJ}	qtf'	b	k_1	k	k_3	Section
Robertson et al.	$W^{RSJ}(t_i)$	$\frac{(k_3+1) \cdot qtf(t_i)}{k_3+qtf(t_i)}$	∈[0.6,0.75]	∈[1.0,2.8]	k_1	8	2.2.1
Rasolofo and Savoy	$max\{0, \log \frac{N-df(t_i)}{df(t_i)}\}$	$\frac{qtf(t_i)}{k_3+qtf(t_i)}$	0.9	1.2	2	1,000	2.4.1
Uematsu et al.	$max\{0, \log \frac{N-df(t_i)+0.5}{df(t_i)+0.5}\}$	1	0.75	1.2	1.2	none	2.4.3
Büttcher et al.	$\log \frac{N}{df(t_i)}$	1	0.5	1.2	1.2	none	2.4.2
Tao and Zhai	$max\{0, \log \frac{N-df(t_i)+0.5}{df(t_i)+0.5}\}$	$\frac{(k_3+1) \cdot qtf(t_i)}{k_3+qtf(t_i)}$	optimal	1.2	1.2	1,000	2.4.5
Cummins and O'Riordan	$max\{0, \log \frac{N-df(t_i)+0.5}{df(t_i)+0.5}\}$	$qtf(t_i)$	0.75	0	1.2	none	2.7.4

Table 2.1: Overview: BM25 variations.

In Tao and Zhai [TZ07], b is tuned for optimality on BM25, although its exact value is not reported. For our own experiments, we use the BM25 score as used by Büttcher et al. in [BC05, BCL06].

2.2.2 Lnu.ltc

Buckley et al. [BSM95] introduced the *Lnu.ltc* weighting scheme used by Monz [Mon04] (cf. Section 2.4.4) in a normalized version. *Lnu* specifies the document weight that is determined by a logarithmically smoothed term frequency and a pivoted length normalization. *Ltc* relates to the query term weight that is computed by a logarithmically smoothed query term frequency in combination with *idf* and a standard cosine normalization.

[2]The original BM25 paper [RW94] does not specify the parameter choices while [RWHB+95] shows typical values and ranges.

The formulation for the *Lnu* weighting is

$$lnu(d,t_i) = \frac{\frac{1+\log(tf(t_i,d))}{1+\log(avg_{t_j \in \{t:tf(t,d)>0\}} tf(t_j,d))}}{(1-slope)\cdot pivot + slope \cdot dt(d)}.$$

The *slope* value is fixed at 0.2, *pivot* is set to the average number of distinct terms per document in the collection. The *ltc* weighting scheme for queries is

$$ltc(t_i) = \frac{(\log(qtf(t_i))+1)\cdot idf_1(t_i)}{\sqrt{\sum_{t_x\in q}[(\log(qtf(t_x))+1)\cdot idf_1(t_x)]^2}}.$$

2.2.3 ES

Cummins and O'Riordan employ a term weighting scheme learned in [CO07]

$$score_{ES}(d,q) = \sum_{t_i \in T_d(P_d(q))} \frac{tf(t_i,d)\cdot qtf(t_i)}{tf(t_i,d) + 0.45\cdot\sqrt{\frac{l_d}{avgdl}}} \cdot \sqrt{\frac{ctf(t_i)^3 \cdot N}{df(t_i)^4}}$$

and linearly combine it with their proximity score combinations learned by Genetic Programming as described in [CO09]. Section 2.7.4 elaborates on the details of the learning process and specifies the learned proximity scores.

2.2.4 Language Models, Smoothing Methods, and KL-Divergence

Another group of models are language models (LM) which have been employed in several areas of computer science such as speech recognition. Ponte and Croft [PC98] and Hiemstra [Hie98] were the first to use language models in Information Retrieval. Language models aim at modelling the query generation process. To this end, they rank documents according to the likelihood that a random sample of a document generates a given ordered query $S_q = (t_1,\dots,t_n)$. This likelihood is captured by means of a document language model for each document. The most basic language model is the unigram language model that uses bag-of-words. It relies only on term distributions and does not use any context information:

$$P_{unigram}(q|d) = \prod_{i=1}^{n} P(t_i|d),$$

where $P(t_i|d) = \frac{tf(t_i,d)}{l_d}$ which corresponds to the maximum likelihood model for document-term probability.

For completeness, we introduce bigram language models which consider the previous term as a context (and therefore already incorporate some proximity information by means of the context information) such that

$$P_{bigram}(q|d) = P(t_1|d)\prod_{i=2}^{n} P(t_i|t_{i-1},d),$$

where $P(t_i|t_{i-1},d) = \frac{tf((t_{i-1},t_i),d)}{tf(t_{i-1},d)}$. The general form of n-gram language models considers the previous n-1 terms and defines probabilities analogously to the bigram language model.

In [ZL04], Zhai and Lafferty survey different smoothing methods and compare their performance. Smoothing methods aim at adapting the maximum likelihood estimator such that data sparseness is compensated. Jelinek-Mercer smoothing [JM80] uses a linear interpolation of the maximum likelihood model for document-term probability as foreground model and the collection-term model as background model. It uses a mixture parameter δ to control the influence of each model

$$P(t|d,\mathcal{C}) = (1-\delta)\cdot P(t|d) + \delta \cdot P(t|\mathcal{C}),$$

where $P(t|d) = \frac{tf(t,d)}{l_d}$ and $P(t|\mathcal{C}) = \frac{ctf(t)}{l_\mathcal{C}}$. For a non-seen term t in d, $tf(t,d)$ is 0 which would make the score of that document zero for any query containing t (as $P(t|d) = 0$); smoothing aims at fixing that flaw by introducing the background model. Another popular smoothing method surveyed in [ZL04] is the Dirichlet prior

$$P(t|d,\mathcal{C}) = \frac{tf(t,d) + \mu \cdot P(t|\mathcal{C})}{l_d + \mu},$$

where μ is a smoothing parameter.

KL-divergence [LZ01] measures the difference of two probability distributions. In the case of language models it compares a query language model and a document language model. The basic form of the KL-divergence model is defined as

$$KL(f,g) = \sum_x f(x) \cdot log\frac{f(x)}{g(x)}.$$

If f and g represent the same distribution, their KL-divergence value becomes 0 – for larger values, the divergence is larger. Lv and Zhai (cf. Section 2.6.1) use KL-divergence to compare the similarity between a query language model and their positional language model that constructs a language model at each term position. The KL-divergence language model variant used by Tao and Zhai is defined as

$$score_{KL}(d,q) = \sum_{t \in T_d(P_d(q))} (qtf(t) \cdot ln(1 + \frac{tf(t,d)}{\mu \cdot p(t|\mathcal{C})})) + |q| \cdot ln\frac{\mu}{l_d + \mu},$$

where $p(t|\mathcal{C}) = \frac{ctf(t)}{l_\mathcal{C}}$.

2.3 Example

As a running example, we will use a poem written by Amy Lowell (taken from "A Dome of Many-Coloured Glass") which is depicted in Figure 2.2. Superscripts represent term positions. Our query will be $q = \{sea, shell, song\}$ or, for order-aware scoring models, $S_q = (sea, shell, song)$.

The query terms (with position information, disabling match cases and ignoring punctuation) in the poem are located at $\{sea^1$, $shell^2$, sea^3, $shell^4$, sea^5, $shell^6$, $song^{10}$, $song^{14}$, sea^{53}, $shell^{54}$, sea^{55}, $shell^{56}\}$.

Sea^{1} Shell^{2}
Sea^{3} Shell,^{4} Sea^{5} Shell,^{6}
Sing^{7} me^{8} a^{9} song,^{10} O^{11} Please!^{12}
A^{13} song^{14} of^{15} ships,^{16} and^{17} sailor^{18} men,^{19}
And^{20} parrots^{21}, and^{22} tropical^{23} trees,^{24}
Of^{25} islands^{26} lost^{27} in^{28} the^{29} Spanish^{30} Main^{31}
Which^{32} no^{33} man^{34} ever^{35} may^{36} find^{37} again,^{38}
Of^{39} fishes^{40} and^{41} corals^{42} under^{43} the^{44} waves,^{45}
And^{46} seahorses^{47} stabled^{48} in^{49} great^{50} green^{51} caves.^{52}
Sea^{53} Shell,^{54} Sea^{55} Shell,^{56}
Sing^{57} of^{58} the^{59} things^{60} you^{61} know^{62} so^{63} well.^{64}

Figure 2.2: A poem with position information.

2.4 Linear Combinations of Scoring Models

One category of text-proximity enhanced scoring models is based on linearly combining content and proximity scores. Such scoring models always attribute a relevance score of the following form to a given document d with respect to a query q:

$$score(d,q) = \lambda \cdot cscore(d,q) + (1-\lambda) \cdot pscore(d,q), \lambda \in (0,1).$$

While *cscore* denotes the content score, *pscore* denotes the proximity score. In this section, we will present several approaches that can be assigned to this class of scoring models, namely scoring approaches by Rasolofo and Savoy [RS03], Büttcher et al. [BC05, BCL06], Uematsu et al. [UIF^{+}08], Monz [Mon04], and Tao and Zhai [TZ07]. Please note that the absolute scores computed in [RS03, BC05, BCL06, Mon04, UIF^{+}08, TZ07] differ by a factor of two from the descriptions presented here to fit our framework. Dividing the scores from the original papers by two however neither influences the ranking (as the order of scores is preserved) nor the ratio between scores attributed to documents.

2.4.1 Rasolofo and Savoy

Rasolofo and Savoy [RS03] compute results of a query q by means of a two-stage algorithm: in stage one, the algorithm computes the top-100 documents from $\mathcal{C}$ according to the *cscore* which is a variant of Okapi BM25 described in Section 2.2.1. In stage two, it reranks these documents. To this end, for every such document, it computes the *pscore*. Reranking just the top-100 documents from stage one is motivated by efficiency needs and the main interest to improve the ranking of the top-ranked documents.

The algorithm sequentially reads the query term positions within d and computes a weight for each pair of query term positions $(i,j) \in Q_{all,d}(q, dist)$ as

$$tpi(i,j) = \frac{1}{(i-j)^2}.$$

The underlying assumption is that there is no semantic relationship between two keywords located in a text window with a width that exceeds *dist*. [RS03] sets *dist* to a value of 5. This means for the poem example and *dist*=5 that $song^{10}$ only influences (and is influenced by) sea^5, $shell^6$, and $song^{14}$ as these three occurrences are located within the text window of $song^{10}$. We define the sum of *tpi* contributions of term pair (t_i, t_j) within a text window of size *dist* as

$$tpiacc_d(t_i, t_j, dist) = \sum_{(i,j)\in Q_{all,d}(q,dist):p_i=t_i\wedge p_j=t_j} tpi(i, j).$$

As $shell^4$ or sea^{53} are too distant from $song^{10}$ in document d (i.e., they are not part of $song^{10}$'s text window), the term pairs ($shell^4$, $song^{10}$) and ($song^{10}$, sea^{53}) do not influence $tpiacc_d(shell, song, 5)$ and $tpiacc_d(sea, song, 5)$, respectively.

For our example the formula leads to the following *tpiacc* scores:

$$\begin{aligned}
tpiacc_d(sea, sea, 5) =& \frac{1}{(3-1)^2} + \frac{1}{(5-1)^2} + \frac{1}{(5-3)^2} + \frac{1}{(55-53)^2} = 0.625,\\
tpiacc_d(sea, shell, 5) =& \frac{1}{(2-1)^2} + \frac{1}{(4-1)^2} + \frac{1}{(6-1)^2} + \frac{1}{(3-2)^2} + \frac{1}{(5-2)^2} +\\
& \frac{1}{(4-3)^2} + \frac{1}{(6-3)^2} + \frac{1}{(5-4)^2} + \frac{1}{(6-5)^2} + \frac{1}{(54-53)^2} +\\
& \frac{1}{(56-53)^2} + \frac{1}{(55-54)^2} + \frac{1}{(56-55)^2} = 8.484,\\
tpiacc_d(sea, song, 5) =& \frac{1}{(10-5)^2} = 0.04,\\
tpiacc_d(shell, shell, 5) =& \frac{1}{(4-2)^2} + \frac{1}{(6-2)^2} + \frac{1}{(6-4)^2} + \frac{1}{(56-54)^2} = 0.8125,\\
tpiacc_d(shell, song, 5) =& \frac{1}{(10-6)^2} = 0.0625, \text{and}\\
tpiacc_d(song, song, 5) =& \frac{1}{(14-10)^2} = 0.0625.
\end{aligned}$$

The weight for a pair of query terms (t_i, t_j)

$$w_d(t_i, t_j, dist) = \frac{tpiacc_d(t_i, t_j, dist) \cdot (k_1 + 1)}{tpiacc_d(t_i, t_j, dist) + K}$$

is structure-wise similar to the term frequency component of BM25, substituting $tf(t, d)$ for $tpiacc_d(t_i, t_j)$. Finally, the proximity scoring function for a document d on query q sums up the contributions of all pairs of query terms in document d. Hence, the formulation

$$pscore(d, q, dist) = \sum_{(t_i,t_j)\in q\times q} w_d(t_i, t_j, dist) \cdot \min\{qw(t_i), qw(t_j)\},$$

where $qw(t_i) = idf_2(t_i) \cdot \frac{qtf(t_i)}{k_3+qtf(t_i)}$, which shrinks the influence of a query term to the importance of the least important term in the considered pair. The final score is defined as

$$score_{\text{Rasolofo}}(d, q, dist) = \frac{1}{2} \cdot cscore(d, q) + \frac{1}{2} \cdot pscore(d, q).$$

2.4.2 Büttcher et al.

Büttcher et al. [BC05, BCL06] combine the baseline BM25 scoring function[3] with a proximity score which we will describe in the following to compute document-level relevance scores. For any document d, they maintain for every query term t_k an accumulator value denoted by $acc_d(t_k)$. This accumulator value can be summarized by the following formula:

$$acc_d(t_k) \quad = \sum_{(i,j)\in Q_{adj,d}(q):p_i\neq p_j,p_i=t_k} \frac{idf_1(p_j)}{(j-i)^2} + \sum_{(i,j)\in Q_{adj,d}(q):p_i\neq p_j,p_j=t_k} \frac{idf_1(p_i)}{(j-i)^2}.$$

Büttcher et al. use adjacent query term occurrences to compute accumulator values. Adjacency is used in the broader sense here such that non-query terms might be located between adjacent query terms. It is obvious that the accumulator value increases the more, the less distant the occurrences of two distinct terms are and the less documents in the collection contain the adjacent term.

For our example we demonstrate how to compute acc_d scores. To this end we have to consider $Q_{adj,d}(q)= \{(1,2),(2,3),(3,4),(4,5),(5,6),(6,10),(10,14),(14,53),(53,54),(54,55),(55,56)\}$ which contains the position information of adjacent query term occurrences in the example poem.

$acc_d(t)$ considers *idf* scores of t's adjacent query terms. We briefly explain how to compute $acc_d(song)$: for this purpose we consider all query term occurrences adjacent to any occurrence of query term *song* in d. $song^{14}$ is adjacent to sea^{53} and $shell^{6}$ is adjacent to $song^{10}$. $acc_d(song)$ is increased by the *idf* scores of the adjacent terms of *'song'* but decreases with the square of increasing distance to the adjacent terms. Please note that $acc_d(song)$ is not influenced by $(song^{10}, song^{14})$ as $p_{10} = p_{14} = song$. Consequently,

$$acc_d(song) = [\frac{idf_1(sea)}{(53-14)^2}] + [\frac{idf_1(shell)}{(10-6)^2}].$$

The proximity score structurally resembles the BM25 scoring model presented in Section 2.2, substituting the accumulator values for the *tf* values:

$$pscore(d,q) = \sum_{t\in q} min\{1, idf_1(t)\} \frac{acc_d(t)\cdot(k_1+1)}{acc_d(t)+K}.$$

The document score for a document d structurally corresponds to the one formulated in Subsection 2.4.1:

$$score_{\text{Büttcher}}(d,q) = \frac{1}{2}\cdot cscore(d,q) + \frac{1}{2}\cdot pscore(d,q).$$

In Büttcher et al.'s approach [BC05, BCL06] only adjacent query terms of the same document influence a query term's aggregated proximity score. This varies the previous work by Rasolofo and Savoy [RS03] that considers all query terms within a given text window. Moreover, Büttcher et al. limit the term proximity score's influence on the document score for terms occurring in just a few documents. They do so by restricting $idf_1(t)$ as a multiplier in *pscore* to one.

[3] with $b = 0.5$ and $k = k_1 = 1.2$

2.4.3 Uematsu et al.

From a structural point of view, Uematsu et al.'s approach [UIF+08] is very similar to Büttcher et al.'s approach. Like Büttcher et al., they use a variant of BM25 (in a slightly different version) as *cscore*. Details can be found in Table 2.1.

The proximity score structurally resembles the *cscore*, however substitutes the *tf* values by co-occurrence values of all query terms. Here $co_{occ}(d,q)$ counts the number of sentences where **all** query terms from q occur:

$$pscore(d,q) = \sum_{t_i \in q} \frac{co_{occ}(d,q) \cdot (k_1 + 1)}{co_{occ}(d,q) + K} \cdot idf_3(t_i).$$

While the first sentence (positions 1 to 12) in our running example contains all query terms (*sea*, *shell*, and *song*) at least once, the second sentence (positions 13 to 52) only contains the query term *song* and the third sentence (positions 53 to 64) only *sea* and *shell*, but not *song*; hence $co_{occ}(d,q)$=1. $score_{\text{Uematsu}}(d,q)$ combines its *cscore* and *pscore* in the same way as $score_{\text{Büttcher}}$:

$$score_{\text{Uematsu}}(d,q) = \frac{1}{2} \cdot cscore(d,q) + \frac{1}{2} \cdot pscore(d,q).$$

2.4.4 Monz

Monz [Mon04] uses a normalized version of Buckley's *Lnu.ltc* weighting scheme [BSM95] as *cscore*. Monz normalizes the *Lnu.ltc* score

$$score_{Lnu.ltc}(d,q) = \sum_{t_i \in q} lnu(d,t_i) \cdot ltc(t_i)$$

with respect to the maximal similarity score of the query such that

$$score_{Lnu.ltc,norm}(d,q) = \frac{score_{Lnu.ltc}(d,q)}{\max_{d \in \mathcal{C}} score_{Lnu.ltc}(d,q)}.$$

The model used for the *pscore* builds on the concept of a minimal matching span, which is the smallest text excerpt that contains all terms that occur both in the query and in the document. To capture the minimal matching span more formally, Monz defines the concept of matching spans. Given a document d and a query q, a *matching span ms* is a set of consecutive positions, where $q \cap T_d(\{1,\ldots,l_d\}) = q \cap T_d(ms)$. That means the consecutive document part represented by ms contains every query term that occurs in document d at least once. The length of a matching span is defined as

$$length(ms) = max(ms) - min(ms) + 1,$$

$max(ms)$, $min(ms)$ being the highest and lowest position in ms, respectively.

A matching span for d and q with the shortest length is called minimal matching span $mms(d,q)$; its length is denoted $length(mms(d,q))$. If multiple minimal matching spans exist, we can safely pick any of them (e.g., the one with the lowest $min(ms)$)

to compute the proximity score *pscore*: we can do that since the *pscore* only uses $length(mms(d,q))$ (multiple minimal matching spans have the same length) and the number of query terms in the minimal matching span (which is also equal for multiple matching spans since they contain all query terms that occur in the document).

We illustrate the concepts of matching span and minimal matching span using our running example, showing again the query term occurrences in d:

$$sea^1, shell^2, sea^3, shell^4, \underbrace{sea^5, shell^6, song^{10}}_{mms(d,q)}, \underbrace{song^{14}, sea^{53}, shell^{54}}_{ms}, sea^{55}, shell^{56}.$$

Matching spans of d contain all terms that occur both in q and d. Therefore, $\{14, \ldots, 54\}$, but also others like $\{14, \ldots, 56\}$ qualify as matching spans. A matching span with the smallest length, called minimal matching span, however, consists of $\{5, \ldots, 10\}$. Note that in [Mon04] minimal matching spans have been defined ambiguously. Instead of defining the minimal matching span as the matching span with the shortest length in the given document, it has only been checked that a given matching span does not contain another matching span with lower length as a subset. However, this might result in multiple minimal matching spans of different lengths: in the example given above, employing the ambiguous definition of Monz [Mon04], $\{14, \ldots, 54\}$ and $\{6, \ldots, 53\}$ would qualify as minimal matching spans besides $\{5, \ldots, 10\}$ since all of them do not contain another matching span with lower length as subset.

The span size ratio considers the proximity of matching terms and is defined as

$$ssr(d,q) = \frac{|q \cap T_d(\{1, \ldots, l_d\})|}{length(mms(d,q))}.$$

It measures how large the document excerpt has to be in order to cover all possible distinct query terms in a document. In our example, $length(mms(d,q))$ = 10 - 5 + 1 = 6 and $ssr(d,q) = \frac{3}{6} = 0.5$. The matching term ratio

$$mtr(d,q) = \frac{|q \cap T_d(\{1, \ldots, l_d\})|}{|q|}$$

measures the fraction of covered query terms in a document which is $\frac{3}{3} = 1$.

Span size ratio and matching term ratio are used to compute the

$$pscore(d,q) = ssr(d,q)^{\alpha} \cdot mtr(d,q)^{\beta}.$$

Here, α and β are additional weights for the span size ratio and the matching term ratio, respectively.[4] The score of a document is then computed as

$$score_{Monz}(d,q) = \begin{cases} \lambda \cdot cscore(d,q) + (1-\lambda) \cdot pscore(d,q) : |q \cap T_d(\{1, \ldots, l_d\})| > 1 \\ cscore(d,q) : else \end{cases}.$$

If d contains only one query term (i.e., $|q \cap T_d(\{1, \ldots, l_d\})| = 1$), the *pscore* is omitted. If $|q \cap T_d(\{1, \ldots, l_d\})| > 1$ (i.e., the document d and query q have more than one query term in common), both *cscore* and *pscore* influence the final score.

[4]Monz uses α=0.125, β=1.0, and λ=0.4 for his experiments.

2.4.5 Tao and Zhai

Tao and Zhai [TZ07] linearly combine a baseline *cscore* with a proximity score. The baseline scores are 1) the KL-divergence model and 2) the Okapi BM25 model as described in Section 2.2.

The authors outline five *proximity distance functions* which can be classified into *span-based* and *distance aggregation measures.* The first class computes proximity scores based on the length of a text segment that covers all query terms. The second class aggregates distances over pairs of query terms and is more local than the first one which takes all query terms into account.

The authors use two different *span-based measures*:

1) Span is defined as the length of the document part that covers all query term occurrences in a document, i.e.,

$$Span(d,q) = max(P_d(q)) - min(P_d(q)).$$

2) Min coverage (*MinCover*) uses the length of the shortest document part that covers each query term at least once in a document, i.e.,

$$MinCover(d,q) = min\{max(P') - min(P') : T_d(P') = T_d(P_d(q))\},$$

where P' is a set of positions in document d.

Both span-based measures are normalized such that

$$Span_{norm}(d,q) = \frac{max(P_d(q)) - min(P_d(q))}{|P_d(q)|} \text{ and}$$

$$MinCover_{norm}(d,q) = \frac{min\{max(P') - min(P')|T_d(P') = T_d(P_d(q))\}}{|T_d(P')|}.$$

Distance aggregation measures come in three variants and are all based on the minimum distance between pairs of query terms t_a and t_b defined as

$$mindist(t_a,t_b,d) = min\{|i-j| : p_i(d) = t_a \wedge p_j(d) = t_b\}.$$

Those three variants encompass

1) Minimum pair distance (*MinDist*) which is the smallest distance over all query term pairs in document d, i.e.,

$$MinDist(d,q) = min_{t_a,t_b \in T_d(P_d(q)), t_a \neq t_b}\{mindist(t_a,t_b,d)\}.$$

2) Average pair distance (*AvgDist*) which is the average distance over all query term pairs in document d, i.e.,

$$AvgDist(d,q) = \frac{2}{n(n-1)} \sum_{t_a,t_b \in T_d(P_d(q)), t_a \neq t_b} mindist(t_a,t_b,d)$$

with n being the number of unique matched query terms in d.

3) Maximum pair distance (*MaxDist*) which is the maximum distance over all query term pairs in document d, i.e.,

$$MaxDist(d,q) = max_{t_a,t_b \in T_d(P_d(q)),t_a \neq t_b}\{mindist(t_a,t_b,d)\}.$$

For the case that document d contains just one kind of query term, *MinDist(d,q)*, *AvgDist(d,q),* and *MaxDist(d,q)* are all defined as l_d.

The authors propose two constraints for a function that transforms the value of a proximity distance function $\delta(d,q)$ into a proximity score $\pi(d,q)$ which is a function of $\delta(d,q)$. While the first constraint (called proximity heuristic) attributes smaller proximity scores to larger $\delta(d,q)$, the second constraint suggests a convex-shaped transformation function that only rewards really close term occurrences. Both constraints lead to the definition of a proximity score

$$\pi(d,q) = log(\alpha + e^{-\delta(d,q)}),$$

where α is a tuning parameter. The baseline retrieval models KL-divergence and BM25 are enriched with the proximity score such that

$$R_1(d,q) = \frac{1}{2} \cdot score_{KL}(d,q) + \frac{1}{2} \cdot \pi(d,q) \text{ and } R_2(d,q) = \frac{1}{2} \cdot score_{\text{BM25}}(d,q) + \frac{1}{2} \cdot \pi(d,q).$$

2.5 Integrated Score Models

Another category of proximity-enhanced score models are integrated score models. Unlike the linear combination models presented in Section 2.4, integrated score models do not linearly combine *cscore* and *pscore* parts, but seek providing a holistic, integrated approach to rank.

2.5.1 De Kretser and Moffat

De Kretser and Moffat [dKM99, dKM04] describe a model that does not make use of Okapi BM25 [RW94, RWHB+95], but relies exclusively on proximity scores of query terms in the text collection. It retrieves the exact point of maximum similarity to the query for any given document, not the document as a whole. The presentation of result snippets can benefit from the knowledge about the exact point of maximum similarity as this opens the option to show only relevant document parts to the user. De Kretser and Moffat's key assumption is that text regions having a high density of query terms are considered as highly important, while isolated query terms in a document are considered as less important. Thus, dense text regions are attributed high scores, while text regions consisting of isolated query terms generate lower scores.

To this end, for each query term t, there is a contribution function c_t which expresses the impact of t, occurring at a position l, on the score for position x. There are three main factors that influence contribution functions: shape, height, and spread.

The *shape* of the contribution function determines the region of influence of each appearance of t in d. De Kretser and Moffat implemented triangle, cosine, circle, and arc functions that are plotted in [dKM99]. Unfortunately, the plots depicted in [dKM99] for arc and circle do not match the formulas. Hence, we added two additional functions we named circle' and arc' that match the plots.

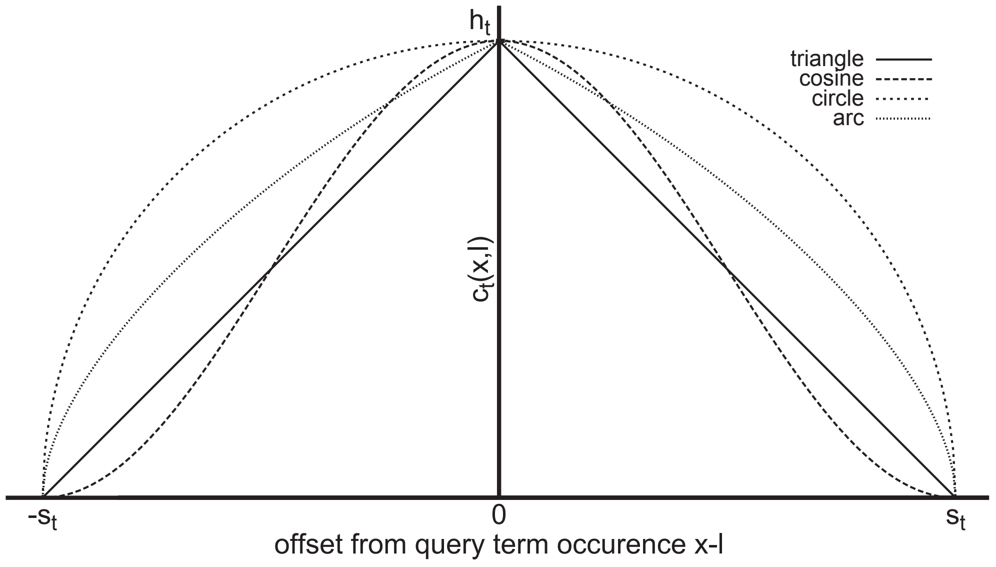


Figure 2.3: Plots according to formulas in [dKM99].

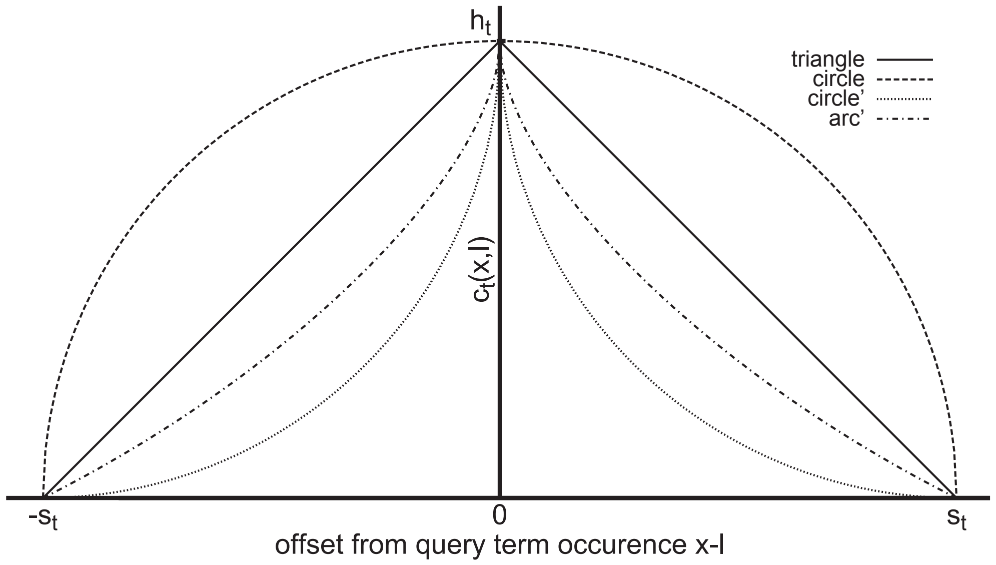


Figure 2.4: Arc and circle replaced to fit the plots in [dKM99, dKM04].

The corresponding contribution functions are listed below:

1. triangle: $c_t(x,l) = max(0, h_t \cdot (1 - \frac{|x-l|}{s_t}))$
2. cosine: $c_t(x,l) = max(0, h_t \cdot \frac{(1+\cos(\pi \cdot \frac{|x-l|}{s_t}))}{2})$
3. circle: $c_t(x,l) = max(0, h_t \cdot \sqrt{1 - (\frac{|x-l|}{s_t})^2})$

4. arc: $c_t(x,l) = max(0, \frac{h_t}{2} \cdot (1 - \frac{|x-l|}{s_t} + \sqrt{1 - (\frac{|x-l|}{s_t})^2}))$
5. circle': $c_t(x,l) = max(0, h_t \cdot (1 - \sqrt{1 - (1 - \frac{|x-l|}{s_t})^2}))$
6. arc': $c_t(x,l) = max(0, \frac{h_t}{2} \cdot (1 - \sqrt{1 - (1 - \frac{|x-l|}{s_t})^2} + (1 - \frac{|x-l|}{s_t}))),$

where $|x - l|$ denotes the positional distance between an occurrence of query term t at position l and the position x we want to compute a score for. Furthermore, h_t represents the height and s_t the spread of t's contribution function. The plots for the functions are depicted in Figure 2.3 and Figure 2.4, respectively.

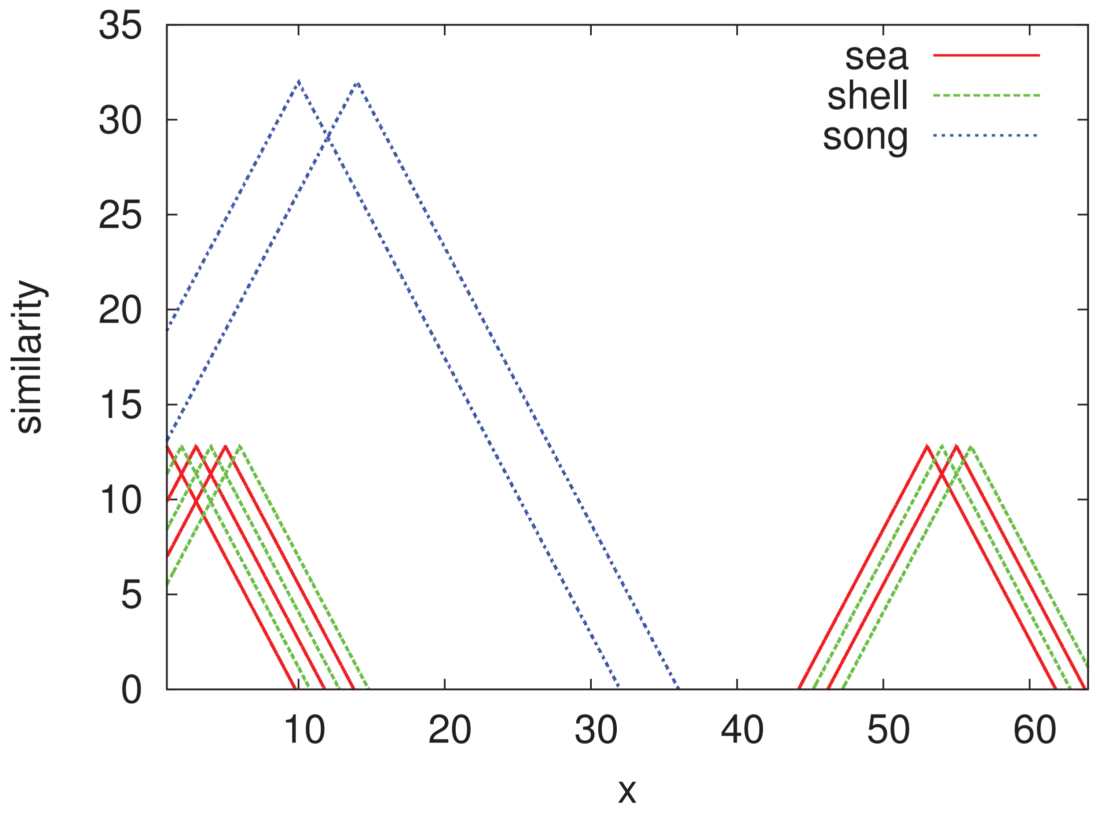


Figure 2.5: Example: triangle-shaped contribution function.

The maximum *height* of the contribution function for a query term occurs at the position of each query term appearance. A query term t generates a contribution function using either a non-damped height

$$h_{t,non-damped} = qtf(t) \cdot \frac{l_{\mathcal{C}}}{ctf(t)}$$

or a damped height

$$h_{t,damped} = qtf(t) \cdot \log_e \frac{l_{\mathcal{C}}}{ctf(t)}$$

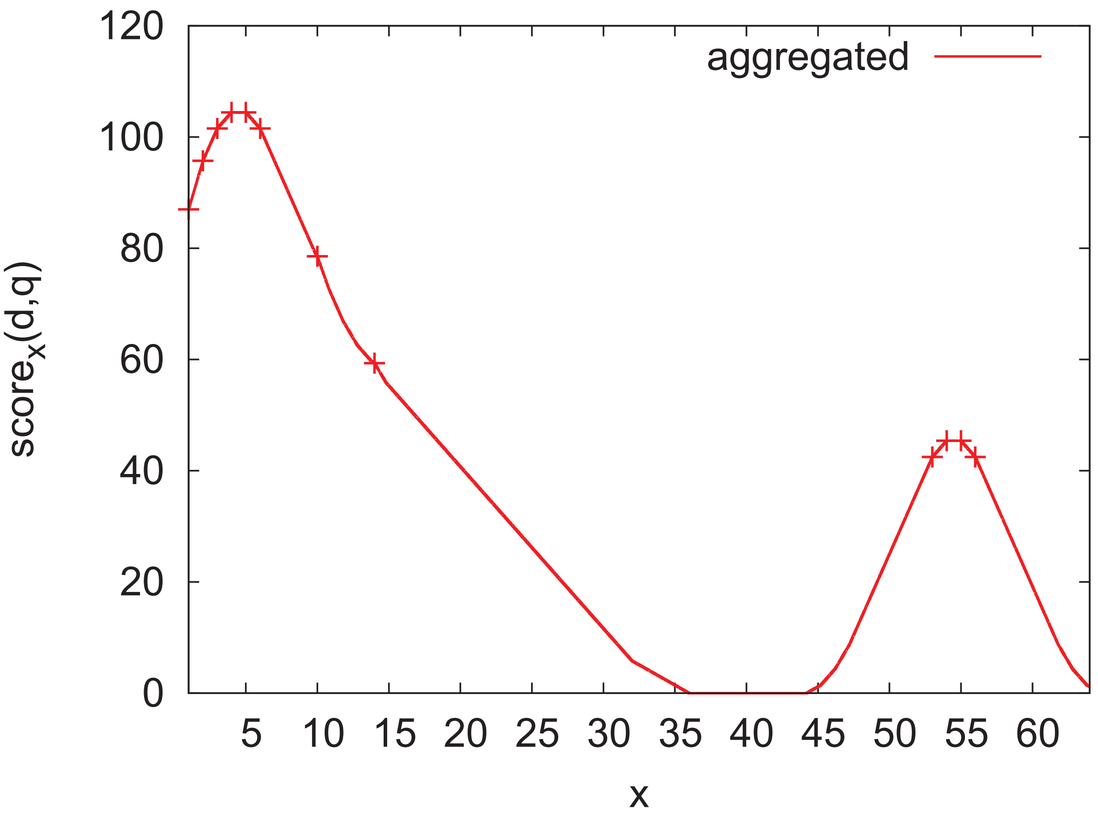


Figure 2.6: Example: aggregated score $score_x$.

which are alternatively used as h_t in the contribution functions. (The usage of *qtf(t)* indicates that this approach allows for term repetitions in the same query.) The *spread or width* of the contribution function determines the distance from the query term appearance in which the query term exerts non-zero influence to the aggregated score. A query term t influences proximity scores of terms within a radius of

$$s_t = \frac{dt}{l_\mathcal{C}} \cdot \frac{l_\mathcal{C}}{ctf(t)} = \frac{dt}{ctf(t)}.$$

The aggregated score for position x in document d and query q is the sum of the contribution function values:

$$score_x(d,q) = \sum_{t \in q} \sum_{l \in P_d(t)} c_t(x,l).$$

Following our running example, Figure 2.5 depicts the individual non-aggregated triangle-shaped contribution functions for each query term occurrence. Figure 2.6 depicts the aggregated scores at all locations in the example document; positions with query term occurrences are marked with crosses.

To reduce computation costs, de Kretser and Moffat restrict the evaluation of aggregated proximity scores to locations where query terms appear. Please note that, for some documents d, the highest $score_x(d, q)$ might be located at a non-query term location x not considered for efficiency reasons. For the example document, whose aggregated scores at various positions are shown in Figure 2.6, this issue does not arise since the highest $score_x(d, q)$ is achieved at positions 4 and 5 where query terms occur.

Figure 2.7 shows a scenario where the highest $score_x(d, q)$ is located at a non-query term position. This example underlies a cosine-shaped contribution function that is applied to an example document that contains the query term *shell* at position 4 and *song* at position 10. The highest $score_x(d, q)$ value, however, is achieved at the non-query term position 6.

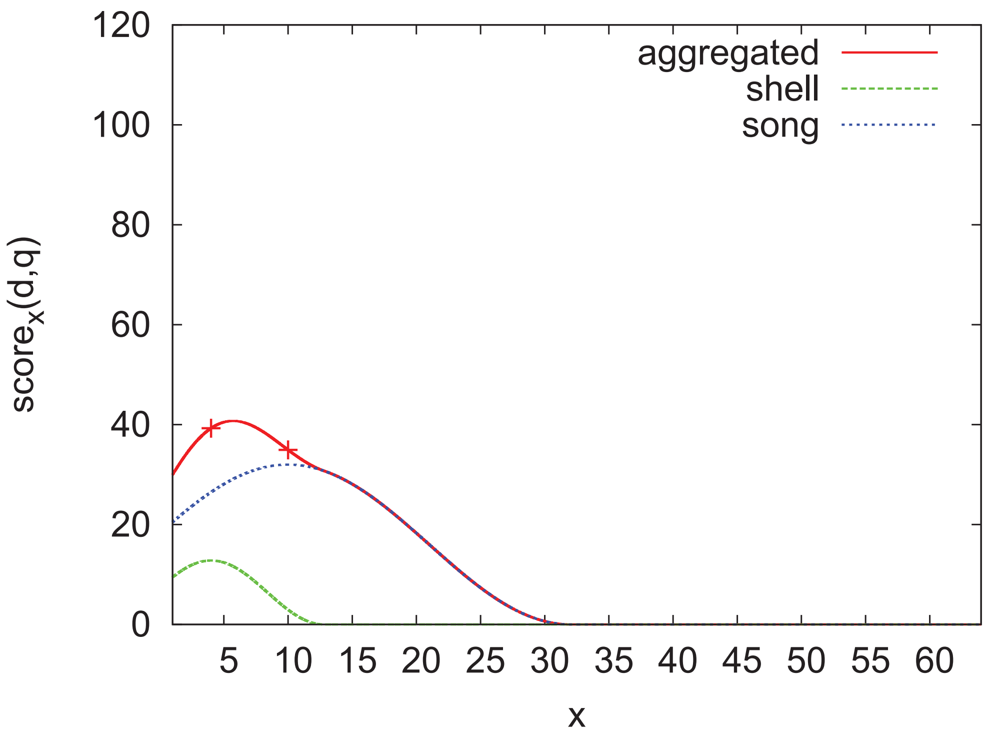


Figure 2.7: Example: highest aggregated score $score_x$ located at a non-query term location.

To obtain a ranking for documents, the authors describe two algorithms: for a given query q, both algorithms start off with retrieving for each document d in the corpus $\mathcal{C}$ the set of positions $P_d(q)$ where query terms occur.

- The first algorithm computes for every document d and all positions $x \in P_d(q)$ the $score_x(d, q)$ at position x in d; the scores from all documents are sorted in descending order. For each document $d \in \mathcal{C}$ the algorithm creates a document accumulator $A[d]$ that keeps the document's score. Now the algorithm starts greedily processing the scores from all documents values and adds them to the corresponding accumulator until k documents have been seen. Those documents are returned as the top-k results.

- The second algorithm computes for each document d the maximum similarity score at any position $x \in P_d(q)$ in d and returns the k documents with the highest scores.

As de Kretser and Moffat consider the first approach more effective, we use this one later for our experiments.

2.5.2 Song et al.

Song et al. [STW$^+$08] describe an algorithm that partitions documents into groups of subsequent query term occurrences. By construction, the query terms in such a group, called *espan* (short for *expanded span*), are pairwise distinct. By means of the espans that contain a query term, the algorithm computes the query term's relevance contribution score (as a substitute for proximity scores) that is directly plugged into an Okapi BM25 ranking function.

The following assumptions underlie the design of the algorithm: the closer appropriately chosen groups of query term occurrences in a document, the more likely that the corresponding document is relevant. The more espans contained in a document, the more likely that the document is relevant. The more query terms an espan of a document contains and the more important these terms are, the more likely that the document is relevant.

The algorithm to detect espans is depicted in Figure 2.8 and proceeds as follows: given a document d and a query q, all query term occurrences form a sequence of (term, position) pairs that are ordered by ascending position; each such pair is called *hit.* We identify the j^{th} query term occurrence in the given document by (a_j, b_j), a_j and b_j being the query term and its position in the document, respectively.

The algorithm distinguishes four cases while scanning the position-ordered sequence of hits:
(1) If the distance between the current hit (a_j, b_j) and the next hit (a_{j+1}, b_{j+1}) is larger than a user-defined threshold d_{max} (i.e., $b_{j+1} - b_j > d_{max}$), a new espan starts with the next hit. This is covered in lines 6-9.
(2) If the current hit (a_j, b_j) and the next hit (a_{j+1}, b_{j+1}) represent the same query term (i.e., $a_j = a_{j+1}$), a new espan starts with the next hit which is described in lines 10-13.
(3) If the next hit (a_{j+1}, b_{j+1}) represents a term a_{j+1} which is identical to a hit's term in the current subchain $currentEspan$, it computes the distance between the current and the next hit as well as the distance between the existing hit and the current hit. The new espan begins at the bigger gap which is handled in lines 14-23.
(4) Otherwise the algorithm scans the next hit in the sequence of (query term, position) pairs which is caught in lines 24 and 25.

Please note that for (3) a tie-breaker is missing, if the distance between the current and the next hit equals the distance between the existing hit and the current hit. For this case our implementation always splits between the current and the next hit.

```
DETECTESPANS(d, q)
    termsAndPositions ← SORTBYPOSITIONASCENDING(d, q)
    length ← termsAndPositions.length()
    espans ← {∅}
    currentEspan ← ∅
    for ($j$ = 1 to length-1)
        if ($(b_{j+1} - b_j) > d_{max}$)
            currentEspan ← currentEspan ∪ {$(a_j, b_j)$}
            espans ← espans ∪ {currentEspan}
            currentEspan ← ∅
        else if ($a_{j+1} = a_j$)
            currentEspan ← currentEspan ∪ {$(a_j, b_j)$}
            espans ← espans ∪ {currentEspan}
            currentEspan ← ∅
        else if ($\exists(a_x, b_x) \in$ currentEspan s.t. $b_{j+1} = b_x$)
            $dist_1 \leftarrow b_{j+1}$ - $b_j$
            $dist_2 \leftarrow b_j$ - $b_x$
            if ($dist_1 \geq dist_2$)
                currentEspan ← currentEspan ∪ {$(a_j, b_j)$}
                espans ← espans ∪ {currentEspan}
                currentEspan ← ∅
            else
                espans ← espans ∪ {currentEspan}
                currentEspan ← {$(a_j, b_j)$}
        else
            currentEspan ← currentEspan ∪ {$(a_j, b_j)$}
    if (length ≠ 0)
        currentEspan ← currentEspan ∪ {$(a_{length}, b_{length})$}
        espans ← espans ∪ {currentEspan}
    return espans
```

Figure 2.8: detectEspans pseudocode.

For a query q, the set of all espans in document d is denoted as $espans(d, q)$. We illustrate now how to compute all espans for a document with the help of our running example, assuming d_{max}=10. The query term occurrences are located at $\{sea^1$, $shell^2$, sea^3, $shell^4$, sea^5, $shell^6$, $song^{10}$, $song^{14}$, sea^{53}, $shell^{54}$, sea^{55}, $shell^{56}\}$.

The first espan consists of $\{sea^1$, $shell^2\}$ by application of (3) since sea^3 is an identical hit to sea^1. As the distance between sea^1 and $shell^2$ equals the distance between $shell^2$ and sea^3, our tie-breaker applies: it splits between the current hit $shell^2$ and the next hit sea^3. The second espan follows the same rule and consists of $\{sea^3$, $shell^4\}$. The next espan consists of $\{sea^5$, $shell^6$, $song^{10}\}$ by application of (2) as $song^{10}$ is identical to $song^{14}$. The distance between $song^{14}$ and sea^{53} exceeds d_{max}. Hence, by application of (1), $\{song^{14}\}$ forms an espan. According to (3) the remaining two espans are $\{sea^{53}, shell^{54}\}$ and $\{sea^{55}, shell^{56}\}$.

Intuitively, for Song et al., the relevance contribution of an espan is a function of its density and the number of query terms occurring in the espan. The density of an espan is defined as

$$density(espan) = \frac{\#\text{query terms}(espan)}{width(espan)},$$

where

$$width(espan) = \begin{cases} \text{maxpos}(espan)\text{-minpos}(espan) + 1\text{: \#query terms}(espan) > 1 \\ d_{max}\text{: else} \end{cases}$$

, maxpos(*espan*)=$max\{b|(a,b) \in espan\}$, and minpos(*espan*)=$min\{b|(a,b) \in espan\}$.

Song et al. measure a term t's relevance contribution given an espan that contains t by means of a function

$$f(t, espan) = (density(espan))^x \cdot (\text{\#query terms}(espan))^y.$$

If the given espan does not contain term t, $f(t, espan)$ is set to zero. For all their experiments Song et al. set x=0.25 and y=0.3, respectively. Depending on the collection, they set b=0.3 and k_1=0.4 (TREC-9 and 10) or b=0.45 and k_1=2.5 (TREC-11).

The relevance contribution of all occurrences of term t in $espans(d, q)$ are accumulated to:

$$rc(t, d) = \sum_{espan_j \in \text{espans}(d,q)} f(t, espan_j).$$

To compute the final score $score_{Song}$, the authors employ Okapi BM25 and replace $tf(t_i, d)$ by $rc(t_i, d)$ with idf_3 as idf score variant such that

$$score_{Song}(d, q) = \sum_{t_i \in q} \frac{rc(t_i, d) \cdot (k_1 + 1)}{rc(t_i, d) + K} \cdot idf_3(t_i).$$

In contrast to Okapi BM25, which attributes a fixed weight of one to each term occurrence, the weight in Song's approach is dependent on the environment of the term occurrence and the density of the espan it has been assigned to.

Although both the approaches proposed by Monz (cf. Section 2.4.4) and Song et al. rely on spans to compute relevance scores, they differ in some features. While Monz considers only the span of minimal length that contains all query terms, Song et al.'s final relevance score incorporates multiple expanded spans. Monz' minimum matching spans contain all query terms that occur in the considered document, Song et al.'s espan may only contain a subset of them. There is a threshold d_{max} that limits the width of expanded spans and the relevance contribution of espans is directly plugged in the Okapi BM25 model.

2.5.3 Mishne and de Rijke

In [MdR05], Mishne and de Rijke make use of a scoring model similar to the tf-idf model [SWY75] and additionally incorporate the coverage of query terms in the document to be scored.

They use the ordered query S_q to construct all possible term-level n-grams that are part of the query following an "everything-is-a-phrase approach". Considering an ordered query S_q=*(sea, shell, song)*, the corresponding 1-grams are *(sea)*, *(shell)* and *(song)*, 2-grams are *(sea, shell)* and *(shell, song)*, while the only 3-gram is *(sea, shell, song)*.

Every term-level n-gram (i.e., n consecutively occurring terms in S_q) derivable from the ordered query S_q forms a *phrase*, with n between 1 and the length of the ordered query.

Proximity terms are term-level n-grams like phrases but the authors use two rewriting methods, namely the *fixed distance* and *variable distance* mode. For *fixed distance proximity terms*, the length of the proximity term n and a tuning parameter k are used as input to a combining method (e.g., $k+n$) that determines the window size where proximity term occurrences in documents are considered (an example follows below). If the distance is $m = k+n$, all term occurrences in a window of size m or less in a document are attributed the same score. For *variable distance proximity terms*, terms that are found in smaller windows than size m in the document are attributed a higher score: window sizes are decreased stepwise from $m = k+n$ to $1+n$ and matching proximity terms are counted in each step. This is equivalent to issuing a query that consists of multiple fixed distance proximity terms of varying size; the tf value of the n-gram can be increased by one for each window size in which the n-gram occurs.

Phrases and proximity terms incorporate position information of query term occurrences into the scoring model and can replace query terms.

The score in its basic form where each t_i represents a query term is defined as

$$\begin{aligned} score_{Mishne}(d, S_q) = \; & \textstyle\sum_{t_i \text{ in } S_q} \frac{\sqrt{qtf(t_i)} \cdot idf(t_i)}{norm(S_q)} \cdot \frac{\sqrt{tf(t_i,d)} \cdot idf(t_i)}{norm(d)} \\ & \cdot mtr(d,q) \cdot weight(t_i), \end{aligned}$$

$$\text{where } norm(S_q) = \sqrt{\sum_{t_i \; in \; S_q} \sqrt{qtf(t_i)} \cdot idf(t_i)^2},$$

$$norm(d) = \sqrt{l_d},$$

$$mtr(d,q) = \frac{|q \cap T_d(\{1,\ldots,l_d\})|}{|q|},$$

$$\text{and } idf(t_i) = 1 + idf_1(t_i).$$

$qtf(t_i)$ counts the number of occurrences of query term t_i in S_q such that for S_q=*(sea, shell, song, sea)*, $qtf(sea)$ would be 2. $weight(t_i)$ is used as a phrase weight proportional to the real term frequency of phrases in different fields of HTML documents (such as BODY, ANCHOR TEXT and TITLE) and seems to be disabled for most evaluation methods. We think that $norm(S_q)$ should be rather $\sqrt{\sum_{t_i \; in \; S_q} \sqrt{qtf(t_i)^2} \cdot idf(t_i)^2}$ to make it appear more similar to a cosine normalization.

The basic form of the scoring model with t_i representing a query term can be varied such that t_i represents a phrase, a fixed distance proximity term or a variable distance proximity term. We illustrate the effects of variable as well as fixed distance proximity terms on the computation of $tf(t_i, d)$ values for the example that t_i represents the 2-gram (*shell, song*).

The query terms *shell* and *song* occur at the following positions in the example document:

$$shell^2, shell^4, shell^6, song^{10}, song^{14}, shell^{54}, shell^{56}.$$

While *(shell, song)* never occurs as a phrase in our example document (the most proximate occurrence of this term pair is $(shell^6, song^{10})$), the term pair can still occur as a proximity term in the document if k is chosen large enough: if k is set to 4 and the combining method is $m = k + n$, a window size of m=6 is induced (as the proximity term *(shell, song)* is a 2-gram which means $n = 2$). Using variable distance proximity terms is equivalent to using some fixed distance proximity terms of varying size; for the example with a window size of $m = 6$, using variable distance proximity terms is equivalent to using four fixed distance proximity terms (with a distance of 6, 5, 4, and 3). As $(shell^6, song^{10})$ has one occurrence in a text window of 5 terms for our example document, in the fixed distance proximity mode it increases $tf((shell, song), d)$ by one, while it increases $tf((shell, song), d)$ by two in the variable distance proximity mode; for a window size of 6 and 5. Being 7 positions apart, $(shell^4, song^{11})$ does not influence $tf(t_i, d) = tf((shell, song), d)$.

For efficiency reasons, document frequencies of phrases and proximity terms are estimated. To estimate the document frequency for a phrase or proximity term p=$(t_y, t_{y+1}, \ldots, t_z)$ with length $|p| = z - y + 1$, Mishne and de Rijke use different heuristics for their estimations of *idf* values:

- `Sum:` $idf(p) = \sum_{i=y}^{z} idf(t_i) = \sum_{i=y}^{z}(1 + idf_1(t_i))$
- `Minimum:` $idf(p) = \min_{i\in\{y,\ldots,z\}} idf(t_i) = \min_{i\in\{y,\ldots,z\}}(1 + idf_1(t_i))$
- `Maximum:` $idf(p) = \max_{i\in\{y,\ldots,z\}} idf(t_i) = \max_{i\in\{y,\ldots,z\}}(1 + idf_1(t_i))$
- `Arithmetic mean:` $idf(p) = \frac{1}{|p|} \cdot \sum_{i=y}^{z} idf(t_i) = \frac{1}{|p|} \cdot \sum_{i=y}^{z} (1 + idf_1(t_i))$
- `Geometric mean:` $idf(p) = \prod_{i=y}^{z} idf(t_i)^{\frac{1}{|p|}} = \prod_{i=y}^{z} (1 + idf_1(t_i))^{\frac{1}{|p|}}$

2.6 Language Models with Proximity Components

This section presents the language models by Lv and Zhai [LZ09] and by Zhao and Yun [ZY09] that exploit proximity information.

2.6.1 Lv and Zhai

In contrast to most other works that deal with language models (LMs), Lv and Zhai [LZ09] do not use one general language model for each document, but one language model for each word position in a document coined positional language model (PLM) estimated based on position-dependent counts of words. In most existing work on LMs, the estimated document language models only consider the word counts in the document, but not the positions of words. PLMs implement two heuristics which are usually treated externally to LM approaches:

1. the proximity heuristic that rewards documents which have closeby occurrences of query terms and
2. passage retrieval that scores documents mainly based on the best matching passage.

PLMs facilitate the optimization of combination parameters that combine proximity and passage retrieval heuristics on one side and language models on the other side. Furthermore, PLMs allow finding best-matching positions in a document, i.e., support soft passage retrieval. PLMs at a position of a document are estimated based on propagated word counts from the words at all other positions in the document: positions closer to a term occurrence in a document get a higher share of the impact than those farther away which captures the proximity heuristics. A similar approach has also been used by de Kretser and Moffat for text retrieval (cf. Section 2.5.1) and by Beigbeder for XML retrieval (cf. Section 5.2.3). The propagation of a term occurrence to other positions is accomplished by a proximity-density function. A PLM is a generalization of a standard document LM and a window passage LM.

Documents can be scored using one PLM or by a combination of multiple PLMs. First, the authors build up a virtual document d_i for each position i in document d. d_i is a term frequency vector whose j^{th} component contains the propagated count $c'(t_j, i)$ of occurrences of term t_j in document d to position i. Thus,

$$p(t|d,i) = \frac{c'(t,i)}{\sum_{t' \in V} c'(t',i)}$$

is a PLM at position i, where

$$c'(t,i) = \sum_{j=1}^{l_d} c(t,j) \cdot k(i,j).$$

$c(t,j)$ is the count of term t at position j which is 1 iff t occurs at j (0 otherwise), and $k(i,j)$ (which can be any non-increasing function of $|i-j|$) serves as a discounting factor. The authors populate the discounting factor with one out of 5 different kernels that determine the influence of a term occurring at position j to position i:

1. Gaussian kernel:
$$k(i,j) = exp[\frac{-(i-j)^2}{2\sigma^2}]$$
2. Triangle kernel:
$$k(i,j) = \begin{cases} 1 - \frac{|i-j|}{\sigma} & \text{if } |i-j| \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$
3. Cosine (Hamming) kernel:
$$k(i,j) = \begin{cases} \frac{1}{2}[1 + cos(\frac{|i-j| \cdot \pi}{\sigma})] & \text{if } |i-j| \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$

4. Circle kernel:
$$k(i,j) = \begin{cases} \sqrt{1-(\frac{|i-j|}{\sigma})^2} & \text{if } |i-j| \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$

5. Passage kernel (=the baseline):
$$k(i,j) = \begin{cases} 1 & \text{if } |i-j| \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$

The spread σ is a tuning parameter which is kept constant for all queries and query terms. De Kretser and Moffat's approach described in Section 2.5.1 makes use of kernel functions named contribution functions; the authors employ triangle, cosine, and circle kernels as used by Lv and Zhai, but also an arc-shaped kernel which is not used here.

Furthermore, Lv and Zhai use two standard smoothing methods, namely Dirichlet prior and Jelinek-Mercer smoothing (adapted to PLMs). Following the descriptions of smoothing methods for LMs in Section 2.2, application of Dirichlet prior smoothing to Lv and Zhai's PLM leads to

$$p_{DP}(t|d,i) = \frac{c'(t,i) + \mu p(t|\mathcal{C})}{(\sum_{t' \in V} c'(t',i)) + \mu},$$

and Jelinek-Mercer smoothing results in

$$p_{JM}(t|d,i) = (1-\lambda)p(t|d,i) + \lambda p(t|\mathcal{C})$$

with $p(t|\mathcal{C}) = \frac{ctf(t)}{l_{\mathcal{C}}}$.

Intuitively, $p(t|d,i)$ describes the share of the impact of t to impacts of all terms at position i in d, the relative influence share of term t to position i in d. For each PLM, the authors adopt the KL divergence model to compute a position i-specific score

$$S(d,q,i) = -\sum_{t \in V} p(t|q) \cdot \log \frac{p(t|q)}{p(t|d,i)},$$

where $p(t|d,i)$ can be either the non-smoothed, Dirichlet prior smoothed ($p_{DP}(t|d,i)$) or Jelinek-Mercer smoothed $p_{JM}(t|d,i)$ variant; $p(t|q)$ is the maximum likelihood estimate (MLE) for a query language model, i.e., $p(t|q) = \frac{qtf(t)}{|q|}$ or a result of a pseudo relevance feedback algorithm.

Ranking options are as follows:

- scoring all documents by the best position in that document:
$$S(d,q) = max_{i \in \{1,\ldots,l_d\}}\{S(d,q,i)\}$$

- scoring all documents by the average of the best k positions in that document:
$$S(d,q) = \frac{1}{k} \cdot \sum_{i \in \text{top-k of all } S(d,q,\cdot)} S(d,q,i)$$

- scoring all documents using a weighted score based on various spreads σ:

$$S(d,q) = \sum_{\sigma \in R} \beta_\sigma \cdot max_{i \in \{1,\ldots,l_d\}} \{S_\sigma(d,q,i)\}$$

with R being a predefined set of spreads and $\sum_{\sigma \in R} \beta_\sigma = 1$.

2.6.2 Zhao and Yun

Zhao and Yun [ZY09] propose a proximity language model that incorporates a so-called proximity centrality and uses Dirichlet prior smoothing. The proximity centrality is computed for every query term and expresses the query term's importance for the proximity structure in document d relative to the query $q = \{t_1, \ldots, t_n\}$.

The score for a document d relative to a query q is defined as

$$score(d,q) = \sum_{tf(t_i,d)>0, t_i \ in \ q} p(t_i|\hat{\theta}_q) \log \frac{p_s(t_i|d,u)}{\alpha_d \cdot p(t_i|\mathcal{C})} + \log \alpha_d,$$

where $\hat{\theta}_q$ represents the language model estimate for q (s.t. $p(t_i|\hat{\theta}_q) = \frac{qtf(t_i)}{|q|}$), and $u = (u_1, \ldots, u_{|V|})$ are hyper-level parameters of the Dirichlet prior with $u_i = \lambda Prox_B(t_i)$.

$$p_s(t_i|d,u) = \hat{\theta}^B_{d,t_i} = \frac{tf(t_i,d) + u_i + \mu p(t_i|\mathcal{C})}{l_d + \sum_{i=1}^{|V|} u_i + \mu}$$

is the seen word probability of t_i in document d wrt its proximity model.

$\alpha_d \cdot p(t_i|\mathcal{C})$ is the probability assigned to unseen words in d, where

$$\alpha_d = \frac{\mu}{l_d + \sum_{i=1}^{|V|} u_i + \mu} \text{ and } p(t_i|\mathcal{C}) = \frac{ctf(t_i)}{l_\mathcal{C}}.$$

The authors implement three variants to compute the proximate centrality $Prox_B(t_i)$ of a term t_i in d:

1. minimum distance: $Prox_{MinDist}(t_i) = f(\min_{t_j \neq t_i, t_j \ in \ q} \{Dis(t_i,t_j,d)\})$

2. average distance: $Prox_{AvgDist}(t_i) = f(\frac{1}{n-1} \sum_{t_j \neq t_i, t_j \ in \ q} Dis(t_i,t_j,d))$,
   where $n = |\{t_j \ in \ q : tf(t_j,d) > 0\}|$

3. summed distance: $Prox_{SumProx}(t_i) = \sum_{t_j \neq t_i, t_j \ in \ q} f(Dis(t_i,t_j,d))$

While $Dis(t_i,t_j,d)$ is the minimum pairwise distance between occurrences of the terms t_i and t_j in document d, f is a non-linear monotonic function to transform a pairwise distance $dist$ into a term proximity score: $f(dist) = x^{-dist}$, where x is a scaling parameter. If not both t_i and t_j occur at least once in d, $Dis(t_i,t_j,d)$ is set to l_d.

2.7 Learning to rank

2.7.1 General Introduction to Learning to Rank Approaches

Learning to rank approaches, a kind of supervised learning approaches, have become popular over the last decade. Supervised learning approaches rely on a training set which consists of a set of training topics, a document collection represented by feature vectors, and the corresponding relevance assessments. According to [Liu11], learning algorithms aim at learning a ranking model (i.e., how to combine the features) such that the learned ranking model can predict the ground-truth labels of the training set as accurately as possible where prediction accuracy is measured using a loss function. [Liu11] contains a comprehensive review of many contributions in the research area of learning to rank.

2.7.2 Svore et al.

Svore et al. [SKK10] extend the work by Song et al. [STW$^+$08] summarized in Section 2.5.2. They provide a measure how to determine the goodness of an espan and extend the espan feature set introduced in [STW$^+$08]. The initial approach presented in [STW$^+$08] used only the density of espans and the number of query terms to assess the goodness of an espan. In this method, the goodness g_s of an espan s is defined as

$$g_s = \sum_{f \in F} \alpha_f v_{f,s},$$

where f is a feature of s taken from a feature set F. α_f denotes the weight of f and $v_{f,s}$ the value of f for s. The goodness score for document d that contains a set of espans S is defined as

$$g_d = \sum_{s \in S} \sum_{f \in F} \alpha_f v_{f,s} = \sum_{f \in F} \alpha_f (\sum_{s \in S} v_{f,s}).$$

The goal is to learn all feature weights α_f. To this end, the sum of the document's espans' feature vectors is input into LambdaRank [BRL06].

Espan-based features used for the goodness score can be assigned to different categories: basic query match features, formatting and linguistic features, and third-party phrase features extracted from Wikipedia titles and popular n-grams from search engine query logs. A detailed list of espan goodness features can be found in Table 2.2.

Model-related features which concern (unigram/standard) BM25, a bigram version of BM25 as well as proximity match features are depicted in Table 2.3. They can be used as additional features to determine the goodness score of a document (substituting $\sum_{s \in S} v_{f,s}$).

2.7.3 Metzler and Croft

In [MC05], Metzler and Croft design a framework to model term dependencies using Markov random fields (MRFs). In statistical machine learning, MRFs are used to

Query Match Features
Espan contains ≥ 2 (≥ 4) query terms (both binary)
Espan length (number of terms in espan)
Count of query terms in espan and density of espan
Formatting and Linguistic Features (F)
Count of indefinite and definite articles in espan
Count of stopwords in espan
Espan contains a sentence (paragraph) boundary (binary)
Espan contains only stopwords (all binary)
Espan contains html markup (bold, italic, tags) (binary)
Third-party Phrase Features (P)
Espan contains an important phrase (binary)
Count and Density of important phrases in espan

Table 2.2: Espan goodness features.

λBM25 Features
Term frequency of query unigrams
Document frequency of query unigrams
Length of body content (number of terms)
λBM25-2 Features
Term frequency of query bigrams
Document frequency of query bigrams
Proximity Match Features
Relevance contribution (per query term, rc in [STW$^+$08])
Number of espans in the document
Maximum, average espan length, maximum, average espan density
Maximum, average count of query matches in espans
Length of espan with highest term frequency
Term frequency of espan with longest length, largest density

Table 2.3: Model feature sets.

model joint distributions. [MC05] models a joint distribution $P_\Lambda(Q, D)$ over random variables for queries Q and documents D (an estimate of the relevance of a document to a query), parameterized by Λ. Λ is estimated given user-defined relevance assessments.

The model uses three kinds of features, namely single query terms, ordered phrases, and unordered phrases. An MRF is generated from an undirected graph G whose nodes represent random variables while edges carry dependence information between random variables. There are two types of nodes, one query node for each query term q_i and one document node D. Dependent query terms are connected to each other by edges. All query term nodes are connected to the document node. There exist three variants of the MRF model:

- Full independence variant (FI): query terms are considered independent given a document D which means that $P(q_i|D, q_{j\neq i}) = P(q_i|D)$, an assumption that many retrieval models like bag-of-words and unigram language models are based on.
- Sequential dependence variant (SD): adjacent query terms are considered dependent; i.e., $P(q_i|D, q_{j\neq i}) = P(q_i|D)$ only for q_j not adjacent to q_i. This variant can represent biterm (non-order-aware occurrences of query terms in documents) and bigram (order-aware occurrences of query terms in documents) models.
- Full dependence variant (FD): all query terms are dependent on each other. The corresponding graph is complete.

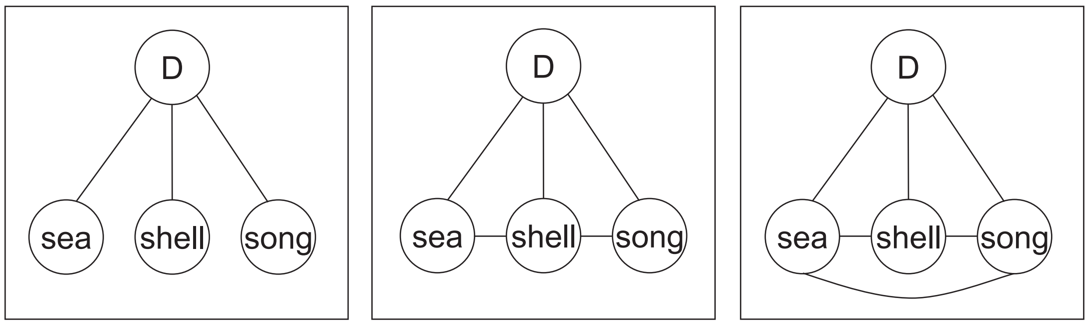


Figure 2.9: Three variants of the MRF model for our running example query, i.e., S_q=*(sea,shell,song)*. We depict (left) the full indepence (FI) variant, (middle) the sequential dependence (SD) variant, (right) the full dependence (FD) variant.

Figure 2.9 illustrates the three variants of the MRF model for the query from our running example which means that S_q=*(sea,shell,song)*. While the FI variant considers the three query terms as independently occurring in documents, the SD variant considers the query as ordered and all adjacent query terms in the query as related: *sea* and *shell* as well as *shell* and *song* are treated as dependent. The FD variant considers all query term pairs as related: *sea* and *shell*, *sea* and *song*, and *shell* and *song* are connected in the graph.

To utilize the MRF model, in a first step, the graph G to represent all query term dependencies is constructed. In a second step, a set of potential functions $\psi(\cdot, \Lambda)$ over cliques in the graph is defined. Potential functions are parameterized as $\psi(c, \Lambda) = exp(\lambda_c f(c))$, where $f(c)$ is a feature function over random variables in a clique c. The joint distribution over the random variables in G is defined as

$$P_\Lambda(Q, D) = \frac{1}{Z_\Lambda} \prod_{c \in C(G)} \psi(c; \Lambda),$$

where $Q = S_q = (q_1, \ldots, q_n)$, $Z_\Lambda = \sum_{Q,D} \prod_{c\in C(G)} \psi(c; \Lambda)$, and $C(G)$ is the set of cliques in G. Metzler and Croft propose three kinds of potential functions that aim at abstracting the idea of term co-occurrence which can be applied to different kinds of cliques:

- 2-clique, one edge between document node D and query node q_i (i.e., $c = q_i, D$):

  $$\begin{aligned}\psi_T(c) &= \lambda_T \log P(q_i|D) \\ &= \lambda_T \log[(1-\alpha_D)\frac{tf(q_i,D)}{l_D} + \alpha_D\frac{ctf(q_i)}{l_{\mathcal{C}}}],\end{aligned}$$

  where $P(q_i|D)$ is a smoothed language modeling estimate which uses a mixture of a document foreground model for document D and a collection background model, $\alpha_D = \frac{\mu}{\mu+l_D}$ the Dirichlet prior (cf. Section 2.6). The potential function measures how likely or well D is described by q_i.

- cliques with two or more query nodes (i.e., $c = q_i, \ldots, q_{i+k}, D$):

  $$\begin{aligned}\psi_O(c) &= \lambda_O \log P(\#1(q_i,\ldots,q_{i+k})|D) \\ &= \lambda_O \log[(1-\alpha_D)\frac{tf_{\#1(q_i,\ldots,q_{i+k}),D}}{l_D} + \alpha_D\frac{ctf_{\#1(q_i,\ldots,q_{i+k})}}{l_{\mathcal{C}}}],\end{aligned}$$

  where $tf_{\#1(q_i,\ldots,q_{i+k}),D}$ is the number of occurrences of the exact phrase $q_i, \ldots, q_{i+k}$ in D.

- an unordered window of size N, cliques with two or more query nodes (i.e., $c = q_i, \ldots, q_j, D$):

  $$\begin{aligned}\psi_U(c) &= \lambda_U \log P(\#uwN(q_i,\ldots,q_j)|D) \\ &= \lambda_U \log[(1-\alpha_D)\frac{tf_{\#uwN(q_i,\ldots,q_j),D}}{l_D} + \alpha_D\frac{ctf_{\#uwN(q_i,\ldots,q_j)}}{l_{\mathcal{C}}}],\end{aligned}$$

  where $tf_{\#uwN(q_i,\ldots,q_j),D}$ is the number of ordered or unordered occurrences of the query terms $q_i, \ldots, q_j$ in D within a window of size N.

In a third step, documents are ranked according to $P_\Lambda(D|Q)$.

$$\begin{aligned}
P_\Lambda(D|Q) &= \frac{P_\Lambda(Q,D)}{P_\Lambda(Q)} \\
&\propto P_\Lambda(Q,D) \\
&= \frac{1}{Z_\Lambda} \prod_{c \in C(G)} \psi(c;\Lambda) \\
&\propto \prod_{c \in C(G)} \psi(c;\Lambda) \\
&\propto \sum_{c \in C(G)} \log \psi(c;\Lambda) \\
&= \sum_{c \in C(G)} \log(\exp(\lambda_c f(c))) \\
&= \sum_{c \in C(G)} \lambda_c f(c) \\
&= \sum_{c \in T} \lambda_T f_T(c) + \sum_{c \in O} \lambda_O f_O(c) + \sum_{c \in O \cup U} \lambda_U f_U(c)
\end{aligned}$$

such that

$$\lambda_T + \lambda_O + \lambda_U = 1.$$

T is the set of 2-cliques representing one query term and a document D, O is the set of cliques with a document node and at least two continuously appearing query terms, and U a set of cliques with a document node and at least two non-contiguously appearing query terms.

λ_T, λ_O, and λ_U need to be tuned such that the retrieval measure for a given test bed is maximized (the authors use the mean average precision value as retrieval measure). As the authors claim that the mean average precision curve has a near concave surface when plotted against the tuning parameters and due to the small number of tuning parameters, this makes tuning by simple hill climbing feasible (i.e., it is unlikely to run into a local maximum value).

2.7.4 Cummins and O'Riordan

Cummins and O'Riordan [CO09] use some term-term proximity measures in a learning to rank framework. To give examples for the various measures, we use our running example, showing again the query term occurrences in d,

$$sea^1, shell^2, sea^3, shell^4, sea^5, shell^6, song^{10}, song^{14}, sea^{53}, shell^{54}, sea^{55}, shell^{56}.$$

For the ease of presentation, we restrict ourselves to the query term pair $(sea, song)$ when we explain term-term proximity measures. Measures which explicitly capture proximity of query term occurrences in documents include

1) the minimum distance between query terms as used in Tao and Zhai (cf. Section 2.4.5):
$$mindist(t_i, t_j, d) = min\{|i-j| : p_i(d) = t_i \wedge p_j(d) = t_j\},$$
where $mindist(sea, song, d) = |5 - 10| = 5$,

2) the distance of average positions of t_i and t_j in d:
$$diff\_avg\_pos(t_i, t_j, d) = |\frac{\sum_{p_i(d)=t_i} i}{tf(t_i, d)} - \frac{\sum_{p_j(d)=t_j} j}{tf(t_j, d)}|,$$
where $diff\_avg\_pos(sea, song, d) = |\frac{1+3+5+53+55}{5} - \frac{10+14}{2}| = |\frac{117}{5} - \frac{24}{2}| = 11.4$,

3) the average distance between all occurrences of t_i and t_j in d:
$$avg\_dist(t_i, t_j, d) = \sum_{i \in P_d(t_i)} \sum_{j \in P_d(t_j)} \frac{|i-j|}{|P_d(t_i)| \cdot |P_d(t_j)|},$$
where $avg\_dist(sea, song, d) = \frac{(9+13)+(7+11)+(5+9)+(43+39)+(45+41)}{5 \cdot 2} = 22.2$,

4) the average of the shortest distance between all occurrences of the least frequently occurring term t_i and any occurrence of the other term t_j:
$$avg\_min\_dist(t_i, t_j, d) = \sum_{i \in P_d(t_i)} \frac{min\{|i-j| : j \in P_d(t_j)\}}{|P_d(t_i)|},$$
where $avg\_min\_dist(sea, song, d) = \frac{(10-5)+(14-5)}{2} = 7$,

5) the smallest average distance $avg\_match\_dist(t_i, t_j, d)$ when each term occurrence has at most one matching distinct term occurrence while there may be two partner term occurrences t_i for some $j \in P_d(t_j)$ in *avg_min_dist*. For *avg_match_dist*, every occurrence of the least frequently occurring term of the term pair in the document has to be paired with a distinct occurrence of the more frequently occurring term of the term pair such that the total distance between the two terms is minimized. To calculate $avg\_match\_dist(sea, song, d)$, either $song^{10}$ or $song^{14}$ can be paired with sea^5, but not both of them; consequently, $avg\_match\_dist(sea, song, d) = \frac{(10-5)+(14-3)}{2} = 8$ or $avg\_match\_dist(sea, song, d) = \frac{(10-3)+(14-5)}{2} = 8$, and

6) the maximum distance between two adjacent occurrences of t_i and t_j,
$$max\_dist(t_i, t_j, d) = max\{j - i : (i, j) \in Q_{adj,d}(\{t_i, t_j\}) \wedge p_i(d) \neq p_j(d)\},$$
where $max\_dist(sea, song, d) = 53 - 14 = 39$.

Another way to implicitly measure proximity uses term frequencies of t_i and t_j in d which comes in the variants

8) $sumtf(t_i, t_j, d) = tf(t_i, d) + tf(t_j, d)$, where $sumtf(sea, song, d) = 5 + 2 = 7$, and

9) $prodtf(t_i, t_j, d) = tf(t_i, d) \cdot tf(t_j, d)$, where $prodtf(sea, song, d) = 5 \cdot 2 = 10$.

High *sumtf* and *prodtf* values increase the probability of closer occurrences for the given term pair (t_i, t_j) in document d.

Other approaches capture information about the entire query (in our example, $q = \{sea, shell, song\}$) by

10) the length of the shortest document part that covers all query term occurrences (corresponds to Tao and Zhai's $Span(d, q)$ measure, cf. Section 2.4.5)

$$FullCover(d, q) = max(P_d(q)) - min(P_d(q))$$

which is $FullCover(d, \{sea, shell, song\}) = 56 - 1 = 55$ in our example, or

11) the length of the shortest document part that covers each query term that occurs in d at least once (also employed by Tao and Zhai as described in Section 2.4.5),

$$MinCover(d, q) = min\{max(P') - min(P') | T_d(P') = T_d(P_d(q))\}.$$

In the example $MinCover(d, \{sea, shell, song\}) = 10 - 5 = 5$.

Normalization measures in use include

12) the length of the document under view l_d, and

13) the number of unique query terms in document d,

$$qt(q, d) = |T_d(\{1, \ldots, l_d\}) \cap q|$$

which is 3 in our example.

Cummins and O'Riordan use genetic programming (GP) to learn good scoring models that combine a subset of the measures presented above. Poli et al. have published a guide to GP [PLM08] that presents an introduction to GP but also advanced techniques in the field. They describe that GP randomly creates an initial population of programs and evolves them from generation to generation using a set of primitive modification operations. All programs are executed and only the best fitting programs per generation survive and are modified using genetic operations to form the candidate set for the next generation. In GP, the primitive modification operations are crossover (i.e., randomly chosen parts of two parent programs are combined), and mutation (i.e., a randomly chosen part of a parent program is randomly changed). When a solution is acceptable or a stopping criterion is reached (e.g., the number of generations exceeds a threshold), the so-far best program is returned as a solution. Solutions are represented using trees. Each tree (genotype) consists of two types of nodes, operators (i.e., functions) or operands (i.e., terminals).

Cummins and O'Riordan run GP six times with an initial population of 2,000 programs for 30 generations and use an elitist strategy which copies the best solution of a generation to the next generation. They employ three constants for scaling ({1,10,0.5}) and seven functions $(+, -, \cdot, /, \sqrt{}, square(), log())$ during evolution with the goal to maximize the MAP metrics performance.

Given an n-term query $\{t_1, \ldots, t_n\}$, the authors represent documents as $n \times n$ matrices, where the diagonal entries are some tf-idf measure $w(t_i)$ per term t_i, and the non-diagonal entries are proximity scores $prox_v(t_i, t_j)$ for pairs of query terms (t_i, t_j), where v denotes the proximity score variant:

$$score(d, q) = \sum_{t_i \in T_d(P_d(q))} \sum_{t_j \in T_d(P_d(q))} \begin{cases} |w(t_i)| & \text{if } i = j \\ |prox_v(t_i, t_j)| & \text{if } i \neq j \end{cases} .$$

The three best proximity functions (from the six runs) are coined $prox_2$, $prox_5$, and $prox_6$ (due to double entries for each query term pair (i.e., there are entries in the document matrix for (t_i, t_j) and (t_j, t_i)), the learned function is twice the value produced by the proximity function):

$$\begin{aligned}
2 \cdot prox_2(t_i, t_j, d) =& log(\frac{10}{mindist(t_i, t_j, d)}) + 5 \cdot \frac{prodtf(t_i, t_j, d)}{avg\_dist(t_i, t_j, d)} + \sqrt{\frac{10}{mindist(t_i, t_j, d)}} \\
2 \cdot prox_5(t_i, t_j, d) =& ((((\frac{log(FullCover(d, q))}{mindist(t_i, t_j, d)^2} + \frac{10}{sumtf(t_i, t_j, d)}) \cdot mindist(t_i, t_j, d) - 0.5) \\
& /mindist(t_i, t_j, d)^2 + \frac{log(0.5) + \frac{prodtf(t_i, t_j, d)}{avg\_dist(t_i, t_j, d)}}{0.5})/mindist(t_i, t_j, d)) - 0.5 \\
2 \cdot prox_6(t_i, t_j, d) =& ((3 \cdot log(\frac{10}{mindist(t_i, t_j, d)}) + log(prodtf(t_i, t_j, d) + \frac{10}{mindist(t_i, t_j, d)}) \\
& + \frac{10}{mindist(t_i, t_j, d)} + \frac{prodtf(t_i, t_j, d)}{sumtf(t_i, t_j, d) \cdot qt(q, d)})/qt(q, d)) \\
& + \frac{prodtf(t_i, t_j, d)}{avg\_dist(t_i, t_j, d) \cdot mindist(t_i, t_j, d)}
\end{aligned}$$

The authors use $score_{ES}$ and a $score_{BM25}$ variant as baselines (cf. Section 2.2 for details); the term weighting scheme $score_{ES}$ (cf. Section 2.2) is linearly combined with the learned proximity score. An additional proximity-enhanced baseline is $score_{ES}$ combined with $MinDist$ as proximity function as used by Tao and Zhai (cf. Section 2.4.5).

2.8 System-Oriented Comparison of Implementation Efforts per Scoring Model

This subsection aims at reviewing the required implementation effort for each of the scoring models we have presented in Chapter 2. Table 2.4 gives just a rough overview of the components needed by each scoring model (with additional remarks in the caption of the table). As content scores which do not use proximity information, BM25 variants, Lnu.ltc, ES, unigram LMs (non-smoothed, Jelinek-Mercer, and Dirichlet prior smoothed), and KL-divergence models do not need materialized term position lists. All presented proximity scoring models can be implemented with term position indexes except Uematsu et al.'s approach [UIF$^+$08] that uses sentence-level term indexes. While all presented linear combination scoring models use *avgdl* and l_d information, most non-linear combination scoring models only use l_d. The presented learning to rank and language model approaches do not incorporate *idf* values, the remaining approaches employ some form of *idf*. ctf values are only used by Metzler and Croft [MC05] as well as de Kretser and Moffat [dKM99, dKM04]: while the first uses ctf for terms, phrases, and unordered windows, the latter uses ctf only for terms. Monz' scoring model [Mon04] is the only model that makes use of the collection-related *dt* and *avgdt* values.

Some variants of the scoring models presented in the original papers may require more features than the ones listed in Table 2.4. We will now provide more details for some scoring models but exclude tuning parameters from our descriptions as we consider them known after training the respective scoring model.

While Rasolofo and Savoy's approach [RS03] considers all query term occurrences in small text windows, Büttcher et al.'s approach [BC05, BCL06] only considers adjacent query term occurrences in unrestricted text windows. Determining those query term occurrences can be implemented using term position lists. For each term, Uematsu et al.'s sentence-level scoring model [UIF$^+$08] needs to index the term occurrences on a sentence-level to compute the number of sentences in a document where **all** query terms co-occur. Monz' approach [Mon04] can be implemented using term position lists that help to determine matching spans and minimal matching spans, respectively. De Kretser and Moffat [dKM99, dKM04] use term positions to determine a score for a given document at a given position: to this end, the positional distances between the scored position and positions of query term occurrences are taken into account.

Song et al.'s approach [STW$^+$08] relies on term position lists to segment documents into espans and uses d_{max} as a maximum width of espans. In some settings, Mishne and de Rijke [MdR05] employ a tag-related weight which is proportional to the number term occurrences within a certain element of an HTML document (e.g., BODY, ANCHOR TEXT or TITLE). Term positions are needed to determine phrases and proximity term occurrences in a complete document or a given tag scope of a document, respectively. Lv and Zhai's approach [LZ09] and Zhao and Yun's approach [ZY09] require term position information to compute kernel values for any position in a document and to compute the proximate centrality of query terms, respectively.

The required implementation effort for the presented learning to rank approaches is highly dependent on the kind of features in use.

Analogously to Song et al.'s approach [STW$^+$08], Svore et al. [SKK10] segment documents into espans using term position lists. The authors can plug a wide choice of different features which influence the required implementation effort in their scoring model. If the model employs formatting features, information about sentence and paragraph boundaries, HTML markup information needs to be stored for each document. If the model uses third party phrase features, it requires lists of important phrases which may not be publicly available. Using λBM25 features requires knowledge about documents' body content lengths, λBM25-2 features require tf and df values for bigrams in documents. The implementation effort for Metzler and Croft's approach [MC05] is dependent on the form of cliques required for scoring. For cliques handling phrases or unordered occurrences of query terms in windows of a given size, one needs to know tf and ctf values for phrases and unordered occurrences of query terms in text windows of a given size, respectively. Deriving those values may be realized using term positions indexes. Materializing tf and ctf values for phrases and unordered occurrences of query terms in text windows is usually only doable for tiny document collections or restricted sets of phrases. Otherwise the required disk space may quickly become prohibitively large.

method	collection-related							document-related				
	$avgtf$	$avgdl$	$avgdt$	$l_{\mathcal{C}},dt$	$idf(t)$	$ctf(t)$	$df(t)$	term pos.	sentence pos.	$tf(t,d)$	l_d	$dt(d)$
BM25		x			idf_x					x	x	
Lnu.ltc (Monz)	x		x		idf_1					x		x
ES(Cummins and O' Riordan)[1]		x				x	x			x	x	
unigram LM										x	x	
Jelinek-Mercer smoothing						x				x	x	
Dirichlet prior smoothing						x				x		
KL-divergence (Tao and Zhai)						x				x	x	
n-gram LM ($n > 1$)								x		$t = n$-gram	x	
Rasolofo and Savoy		x			idf_2			x			x	
Büttcher et al.		x			idf_1			x			x	
Uematsu et al.		x			idf_3				x		x	
Monz								x				
Tao and Zhai(without KL/BM25)								x			x	
de Kretser and Moffat				x		x		x				
Song et al.		x			idf_3			x			x	
Mishne and de Rijke					$1 + idf_1$			x		x	x	
Lv and Zhai[2]								x			x	
Zhao and Yun						x		x		x	x	
Svore et al.[3]								x		x		
Metzler and Croft[4]						x		x		x	x	
Cummins and O' Riordan[5]								x		x	x	

Table 2.4: Overview: Features used in each scoring model. Additional remarks: [1] needs also the number of documents N in the collection, [2] requires ctf and tf if Jelinek-Mercer or Dirichlet prior smoothing are used, [3] determines the set of features dependent on the employed setting (e.g., df and tf for unigrams/bigrams, respectively plus lists of important phrases, etc.), [4] may use tf values not only for terms but also for n-grams and unordered occurrences of n-gram terms, and [5]'s set of features may differ dependent on the learned proximity score.

Chapter 3

Benchmarks

3.1 Introduction

When users look for information, they are driven by an information need. An information need of a user might be: information whether she shall consume black tea or coffee if she suffers from high blood pressure. To represent this information need, users try to formulate queries which usually contain keywords likely to occur in documents that may satisfy the information need, e.g., q={*coffee, black, tea, effect, high, blood, pressure*}. In order to compare the retrieval quality of different search engines that provide result lists as answers to queries that express users' information needs, various initiatives have developed test beds for different application scenarios.

A test bed consists of a document collection, a set of information needs expressed as topics, and a set of relevance assessments which maintain for each topic a list of items judged according to their relevance to the information need. Relevance assessments can be binary-level (i.e., a result is either relevant or non-relevant) or multi-level (e.g., a result is non-relevant, marginally relevant, mostly relevant, or definitely relevant). For classical text retrieval, the granularity of results is typically document-level and the relevance is thus assessed with respect to the complete document while for XML retrieval, results may be parts of the document such as elements or passages whose relevance is also assessed. Relevance is always assessed with respect to the user's information need, not to a query. That means that a result is considered relevant iff it contains some information related to the user's information need. Mere occurrence of the keywords from the query in a result is not sufficient to render the result relevant. Retrieval results are evaluated using various retrieval quality metrics.

In the following, we will describe two popular evaluation initiatives for text and XML retrieval and two less popular, niche representatives for Japanese language and medical search; all of them have in common to provide the means to compare the performance of different systems. After that we will give a detailed description of performance metrics which express the performance of a system under consideration.

3.2 Initiatives

3.2.1 The TREC Initiative and Selected Test Beds

The U.S. NIST (National Institute of Standards and Technology) started their Text REtrieval Conference (TREC) efforts in 1991. The first TREC workshop took place in 1992 and has been run annually since then. It provides a forum for IR researchers to compare their systems to those of others in various areas of IR. A report about the economic impact of the TREC Program can be found at `http://trec.nist.gov/pubs/2010.economic.impact.pdf` and contains much of the information briefly summarized in the following. When NIST started their TREC effort, they aimed at fixing two problems in IR, namely the lack of document collections and of methodologies to enable a standardized comparison of IR systems. They helped to put evaluation efforts into more realistic scenarios by creating many new, large test collections: test collections used for the first TREC in 1992 contained already approx. 750,000 documents compared to a size of 12,000 documents for the largest commonly used collection before. The collection sizes continuously increased to adjust to the growing Web and more powerful machines: in 2004 the GOV2 collection contained approx. 25 million documents (426GB), and the most recent ClueWeb09 collection used for the TREC Web Tracks 2009 and 2010 consisted of more than 1 billion documents (25TB). TREC helped to develop standardized IR evaluation methods by providing document collections, sets of topics and relevance assessments (which documents are relevant to a given query) to compare IR systems in a standardized manner. Test collections are available not only for established tasks such as ad hoc retrieval but also for newer areas such as video retrieval and spam detection. TREC distributes research results and makes them also available to people not participating in TREC. Evaluation techniques and formats used in TREC inspired a number of other workshops and programs.

TREC runs multiple tracks dedicated to particular areas in IR. Past TREC Tracks are listed at `trec.nist.gov/tracks.html` and include the Blog Track (last run in 2010), Cross-Language Track (in 2002), Enterprise Track, Filtering Track (last run in 2002), Genomics Track (last run in 2007), HARD (High Accuracy Retrieval from Documents, last run in 2005), Interactive Track (last run as adjunct to the Web Track 2003), Million Query Track (last run in 2009), Novelty Track (last run in 2004), Question Answering (QA) Track (last run in 2007), Relevance Feedback Track, Robust Retrieval Track (discontinued after 2005), SPAM Track (last run in 2007), Terabyte Track (last run in 2006), Video Track (last run in 2002, starting 2003 there was an independent evaluation named TRECVID with a workshop taking place), and the former Web Track (last run in 2004). In 2011, tracks encompassed the Chemical IR Track, Crowdsourcing Track, Entity Track, Legal Track, Medical Records Track, Microblog Track, Session Track, and a new Web Track (started in 2009).

The Million Query Track used a large number of incompletely judged queries and aimed to find out whether this is better than the traditional TREC pooling approach. The Robust Track used difficult queries and focused on individual topics' retrieval quality rather than optimizing the average effectiveness. The Web Track used a web collection to perform search tasks on it: the Topic Distillation Task tried to find relevant pages desirable for inclusion in a list of key pages. The Large Web Task used 10,000 search log queries from Alta Vista and Electric Monk [Haw00]. The Ad Hoc Task and Small Web Task in TREC-8 used the same topic set to find out how web data differs from Ad Hoc data [VH00]. The Terabyte Track used a significantly larger collection than used for previous TREC evaluations and aimed to find out whether the evaluation scales.

Table 3.1 shows an overview of selected test beds in the context of TREC which include Tracks/Tasks in TREC, a reference to the employed document collection, and the topic sets. We will now describe some TREC collections that are either used in experiments later in this thesis or have been used in the original papers that introduced the methods in Chapter 2.

The TREC45 and TREC45-CR (a.k.a. TREC-8) collection: TREC Disk 4 contains about 30,000 documents (approx. 235MB, *avgdl*=1373.5) from the Congressional Record of the 103rd Congress (**CR**), about 55,000 documents (approx. 395MB, *avgdl*=644.7) from the Federal Register in 1994 (**FR**), and about 210,000 documents (approx. 565MB, *avgdl*=412.7) published in the Financial Times from 1992 to 1994 (**FT**). TREC Disk 5 contains about 130,000 documents provided by the Foreign Broadcast Information Service (approx. 470MB, *avgdl*=543.6) (**FBIS**) and about 130,000 randomly selected Los Angeles Times articles from 1989 and 1990 (approx. 475MB, *avgdl*=526.5) (**LA Times**). The information given here has been taken from `http://www.nist.gov/srd/nistsd22.cfm` (TREC Disk 4) and `http://www.nist.gov/srd/nistsd23.cfm` (TREC Disk 5) where the disks can also be ordered. While the TREC45 collection is approx. 2,140MB in size, the TREC45-CR (a.k.a. TREC-8) collection's size is only about 1.9GB as it does not contain the data from the Congressional Record of the 103rd Congress. Average document lengths for the subcollections have been mentioned in [VH00] and have been computed without term stemming and without stopword removal. According to [VH97] TREC45-CR consists of 528,155 documents at an *avgdl* of 467.42, and after stopword removal of 263.65. While TREC Disks 4 and 5 have been used to process the Ad Hoc Topics in TREC-6, TREC45-CR has been used for the TREC-8 Web Track Ad Hoc Task and the TREC-12 and TREC-13 Robust Track. TREC Disks 4 and 5 (plus the complete TIPSTER collection) have been used in the QA Track in TREC-9 and TREC-10.

The AQUAINT collection: The information presented here can be found at `http://www.ldc.upenn.edu/Catalog/docs/LDC2002T31/`. The AQUAINT collection consists of newswire text data in English from three sources: the Xinhua News Service from China (January 1996–September 2000) (**XIE**), the New York Times News Service (**NYT**), and the Associated Press Worldstream News Service (June 1998–September 2000) (**APW**). All articles are SGML-tagged text data presenting the series of news stories. There is a single DTD available for all data files in the corpus. The corpus was prepared by the Linguistic Data Consortium (LDC) for the AQUAINT Project to be used by NIST for evaluations. The data files are about 3GB in size and contain approx. 375 million words. The AQUAINT collection has been used for the TREC-11 QA Track Main Task, and for the TREC-14 Robust Track.

The TIPSTER collection: The TIPSTER collection (`http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC93T3A`) comes on three disks and contains articles from the Wall Street Journal (1987–1992) (**WSJ87-92**) (173,252 documents, approx. 81M words), Federal Register (1988 and 1989) (**FR88-89**), Associated Press (1988–1990) (**AP88-90**) (242,918 documents, approx. 114M words), Information from the Computer Select disks copyrighted by Ziff-Davis (1989–1992) (**ZIFF89-92**) (approx. 112M words), San Jose Mercury News (1991) (approx. 45M words), U.S. Patents (1983–1991) (250MB in size), and Department of Energy abstracts (**DOE**) (approx. 28M words) . In total it contains approx. 448 million words in over 700,000 documents with 2.1GB of text (`http://www2.parc.com/istl/projects/ia/papers/sg-sigir93/sigir93.html`). While the TIPSTER Disks 1 and 2 have been used to evaluate the Ad Hoc Topics in TREC-1 to TREC-3, TIPSTER Disks 2 and 3 have been used to process the Ad Hoc Topics in TREC-4, TIPSTER Disk 2 (plus TREC Disk 4) to process the Ad Hoc Topics in TREC-5. The complete TIPSTER collection (plus TREC Disks 4 and 5) has been used in the QA Track in TREC-9 and TREC-10.

The VLC2 collection: The VLC2 collection is about 100GB in size and contains 18.5 million web pages which are part of a web crawl from 1997 carried out by the Internet Archive. It has been described in detail in [HCT98] and has been used for the Large Web Task of the TREC-8 Web Track.

The WT2g collection: The WT2g collection is a 2GB sized subset of the larger VLC2 collection and contains 250,000 documents. Details can be found in [HVCB99]. The collection has been used for the Small Web Task of the TREC-8 Web Track.

The WT10g collection: The WT10g collection (`http://ir.dcs.gla.ac.uk/test_collections/wt10g.html`) consists of 1,692,096 English web documents crawled from 11,680 servers and is about 10GB in size. It is the successor of the WT2g collection and contains 171,740 inter-server links (within the collection). Own experiments using the Galago parser have resulted in $avgdl$=599.41 and after stopword removal $avgdl$=393.74. According to [BCH03a], which contains a lot of information about the construction of the WT10g corpus, the corpus was created to perform repeatable retrieval experiments which model web search better than any previously available test collection. It has been used as document collection for the Web Track in TREC-9 and TREC-10.

The .GOV collection: The .GOV collection is a TREC test collection (`http://ir.dcs.gla.ac.uk/test_collections/govinfo.html`) which is a crawl of .gov (U.S. governmental) web sites from early 2002. In total it contains 1,247,753 documents (of which 1,053,372 are HTML files) that have been truncated to a maximum size of 100KB each (reducing the size from 35.3GB to 18.1GB). This collection has been used for the Web Track, Topic Distillation Task from TREC-11 to TREC-13.

The GOV2 collection: The GOV2 collection is a TREC benchmark collection intended for use in the Terabyte Track (`http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`). It was crawled using NIST hardware and network. This crawl from early 2004 of .gov (U.S. governmental) web sites has an uncompressed size of approximately 426GB and consists of 25,205,179 documents (out of which 23,111,957 are HTML, 2,030,339 PDF, 60,176 plain text, 2,253 MS-Word, 454 postscript files) that have been truncated to a maximum size of 256KB each. Our experiments with the Galago parser have resulted in $avgdl$=886.03, and after stopword removal $avgdl$=633.08. This collection which entails 20 times more documents than the .GOV collection has been used for the Terabyte Track, Ad Hoc Task in TREC-13 to TREC-15, and for the Terabyte Track, Efficiency Task in TREC-14.

The ClueWeb09 collection: The ClueWeb09 dataset was created by the Language Technologies Institute at Carnegie Mellon University (CMU) and consists of 1,040,809,705 web pages in 10 languages (`http://boston.lti.cs.cmu.edu/Data/web09-bst/`). It was crawled in January and February 2009 and encompasses 25TB of uncompressed data. The dataset is used by several tracks of the TREC conference. People participating in the TREC Web Track often restrict the collection first to the 503,903,810 English documents (`http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Dataset+Information#Record_Counts`), from which they chose the 50% documents with the smallest probabilities to be spam according to the Waterloo Fusion spam ranking (`http://plg.uwaterloo.ca/~gvcormac/clueweb09spam/`). The resulting document set has an uncompressed size of about 6TB. In our own experiments with the remaining 251,664,804 documents and our own parser, after stemming and stopword removal, the $avgdl$ value was 3021.49. The ClueWeb09 collection has been used as a document collection for the Web Track, Ad Hoc Task in TREC-18 to TREC-20.

3.2.2 INEX and Selected Test Beds

The second initiative we describe is the INitiative for the Evaluation of XML retrieval (INEX) which exists since 2002. It is the leading workshop on XML retrieval and takes place annually. In contrast to TREC where NIST is responsible for providing the test bed, in INEX only the document collection is provided and participants are asked to formulate topics and judge results for relevance.

Like in TREC, the test collections employed in INEX have grown in size over the years to catch up with data growth in the real world, more powerful machines and to pose new challenges to the participants. While the initial INEX IEEE collection from 2002 consisted of about 12,100 articles with 8 million elements at a size of 494MB, and

TREC	Year	Track/Task	Collection	Topics
TREC-1	1992	Ad Hoc Topics	TIPSTER Disks 1+2	51-100
TREC-2	1993	Ad Hoc Topics	TIPSTER Disks 1+2	101-150
TREC-3	1994	Ad Hoc Topics	TIPSTER Disks 1+2	151-200
TREC-4	1995	Ad Hoc Topics	TIPSTER Disks 2+3	201-250
TREC-5	1996	Ad Hoc Topics	TIPSTER Disk 2+ TREC Disk 4	251-300
TREC-6	1997	Ad Hoc Topics	TREC45	301-350
TREC-7	1998	Ad Hoc Topics	TREC45	351-400
TREC-8	1999	Web Track, Ad Hoc and Small Web Topics	Ad Hoc: TREC45-CR SmallWeb: WT2g	401-450
TREC-8	1999	Web Track, Large Web Task	VLC2	20001-30000
TREC-9	2000	Web Track	WT10g	451-500
TREC-10	2001	Web Track, Ad Hoc Topics	WT10g	501-550
TREC-11	2002	Web Track, Topic Distillation Task	GOV	551-600
TREC-12	2003	Web Track, Topic Distillation Task	GOV	TD1-TD50
TREC-13	2004	Web Track, Topic Distillation Task	GOV	75 topics from WT04-1 to WT04-225
TREC-18	2009	Web Track, Ad Hoc Task	ClueWeb09	wt09-1 to wt09-50
TREC-19	2010	Web Track, Ad Hoc Task	ClueWeb09	51-100
TREC-16	2007	Million Query Track	GOV2	1-10000
TREC-12	2003	Robust Track	TREC45-CR	100 topics from 303-650
TREC-13	2004	Robust Track	TREC45-CR	301-450 (Ad Hoc Topics TREC6–TREC8), 601-650 (new topics TREC-12 Robust Track), 651-700 (new topics TREC-13 Robust Track)
TREC-14	2005	Robust Track	AQUAINT	50 topics from 303-689
TREC-13	2004	Terabyte Track, Ad Hoc Task	GOV2	701-750
TREC-14	2005	Terabyte Track, Ad Hoc Task	GOV2	751-800
TREC-15	2006	Terabyte Track, Ad Hoc Task	GOV2	801-850
TREC-14	2005	Terabyte Track, Efficiency Task	GOV2	1-50000
TREC-9	2000	QA Track	TIPSTER+TREC45	201-893
TREC-10	2001	QA Track, Main Task	TIPSTER+TREC45	894-1393
TREC-11	2002	QA Track, Main Task	AQUAINT	1394-1893

Table 3.1: Some TREC test beds

was increased to approx. 16,800 articles, 11 million elements with 764MB in size in 2005, the change to a Wikipedia collection in 2006 brought more than 1.5 million XML documents in 8 languages (out of them about 660,000 English documents) at a size of approx. 10GB. The current Wikipedia collection in use increased the size to more than 50 GB with more than 2.6 million XML documents and 1.4 billion XML elements.

INEX also offers multiple tracks (`https://inex.mmci.uni-saarland.de/`) to their participants. Past tracks include the Heterogeneous Collection (2004–2006), Relevance Feedback (2004–2006), Natural Language (2004–2006), XML Multimedia (2005–2007), Use Case Studies (only in 2006), XML Entity Ranking (2006–2009), Efficiency (2008 and 2009), Book Search (2007-2010, renamed to Book and Social Search in 2011), Ad Hoc Retrieval (2002–2010), and XML Mining (2007–2010) Tracks.

In 2011, tracks encompass the Book and Social Search, Interactive, Relevance Feedback, Data-Centric, Question Answering (QA), Web Service Discovery, and the newly introduced Snippet Retrieval Track which replaces the former Ad Hoc Track. Mostly for the Ad Hoc Track, INEX uses two types of queries: CO (content-only) queries and CAS (content-and-structure) queries. While CO queries are keyword queries without structural information as used in text retrieval, CAS queries impose structural constraints which position keywords into a structural context.

We will now describe the INEX collections used over the years for the Ad Hoc Track.

The INEX IEEE collection 2002–2004 and its extension from 2005: The initial INEX collection from 2002 consisted of 12,107 marked-up articles with 8 million elements, taken from IEEE journals between 1995 and 2002, 494MB in size, and described in [LT07] which we will summarize here. The collection got extended in 2005 by 4,712 new IEEE articles published between 2002 and 2004, resulting in 16,819 articles with 11 million elements and a size of 764MB. A typical article consists of front matter, body, and back matter. The front matter contains metadata (e.g., title, author, publication information, and abstract). The body contains text embedded in its structural information: sections, sub-sections, and sub-sub-sections that start with a title element followed by paragraphs. The content is extended by references (citations, tables, and figures), item lists, and layout (e.g., emphasised, bold text). The back matter contains bibliography and information about the authors. This collection has been used for the Ad Hoc Track which was the only INEX track in 2002 and 2003, and for the Interactive, Relevance Feedback, and Natural Language Track in 2004.

The INEX Wikipedia collection 2006–2008: The INEX Wikipedia collection used for the INEX workshop from 2006 to 2008 consists of 1,535,355 Wikipedia-based XML documents in 8 languages with a total size of about 10GB. The English part consists of 659,388 English documents which are about 4,600MB in size. The average size of an English document is 7,261 bytes, the average depth of a node in an XML document tree is 6.72, and the average number of elements in a document is 161.35. More detailed information can be found in [DG06a] and [DG06b]. As the collection is highly irregular, there is no DTD available for this collection. This collection has been used for the Ad Hoc Track from 2006-2008, the INEX Efficiency Track 2008, and the Entity Ranking Track 2007 and 2008. The English part of this INEX Wikipedia collection with more than 300,000 images at approx. 60GB size has been used for the Multimedia Track in 2007 while the English part with tagged articles and a size of approx. 6GB has been used as Entity corpus for the Entity Ranking Track in 2007 and 2008.

The INEX Wikipedia collection from 2009: This Wikipedia collection has been newly introduced in 2009 (`http://www.mpi-inf.mpg.de/departments/d5/software/inex/`) and was created at Max-Planck-Institute and Saarland University. It consists of 50.7GB XML-ified Wikipedia articles, with 2,666,190 articles (which is four times the number the English articles in the former Wikipedia collection) and 1.4 billion elements. The collection is annotated with the 2008-w40-2 version of YAGO [SSK07]. Parsing the document collection using the Galago parser resulted in *avgdl*=565.83, and after stopword removal in *avgdl*=393.62. In 2009, the INEX Efficiency Track, the Link-the-Wiki Track, and the Question Answering Track have used this collection.

3.2.3 Other Initiatives

The IREX Project

The IREX (Information Retrieval and Extraction Exercise) project is an evaluation project for Information Retrieval and Information Extraction in Japanese. [SI00] reports on this project, briefly summarized below: the project lasted from May 1998 to September 1999 and ended with an IREX workshop held in Tokyo. More information including the data and tools used for the project can be found at `http://nlp.cs.nyu.edu/irex/`. The IREX project had two tasks, namely the Information Retrieval task (IR) and the Named Entity task (NE). We omit the description of the NE task as the evaluation in related work presented in Chapter 4 only deals with the IR task. There were 30 topics in the IR task and participants were asked to submit their top-300 results for each topic. The employed IREX_IR collection consists of about 212,000 Mainichi newspaper articles written in Japanese that were published in 1994 and 1995.

OHSUMED

A description of the OHSUMED test bed is given in [HBLH94] and `http://ir.ohsu.edu/ohsumed/ohsumed.html`, its characteristics are briefly summarized in the following. The OHSUMED test collection is a subset of MEDLINE, a bibliographic database for medical publications maintained by the National Library of Medicine and about 400MB in size. MEDLINE consists of more than 7 million references starting in 1966, and grows by about 250,000 references per year. The OHSUMED collection contains 348,566 references, a subset of 270 medical journals covering the years 1987 to 1991.

The generally short queries contain a brief statement about the patient and an information need. Relevance assessments distinguish two levels of relevance, namely definitely relevant (DR) and definitely or possibly relevant (D+PR). The test bed contains 101 physician-generated queries during patient care with at least one document considered definitely relevant.

3.3 Measures

In order to assess the retrieval quality of search engines, the IR community has developed several measures which can be classified in measures for text/document retrieval and measures for XML retrieval.

Relevance is always assessed with respect to the user's information need, not to a query. That means that a document is considered relevant iff it contains some information related to the user's information need. Mere occurrence of the keywords from the query in a retrieved document is not sufficient to render the document relevant.

3.3.1 Measures for Text/Document Retrieval

Query processing in a search engine returns a ranked list of results that are assessed using various measures; two of the most prominent measures are precision and recall.

For these measures, the order of (the first n) entries in the ranked list of query results does not influence the value of the measure such that the ranked list can also be viewed as a result set.

Definition 3.3.1. (**Precision at rank n**) Given a set of the first n items $I_n = \{i_1, \ldots, i_n\}$ retrieved as answer to query q, and the set of items R considered relevant to q, precision at rank n is defined as

$$P@n = \frac{|R \cap I_n|}{|I_n|}$$

and measures which fraction of the retrieved items in I_n is actually relevant to the query.

Definition 3.3.2. (**R-Precision**) Given the set of items R considered relevant to a query q, and a set of retrieved items $I_{|R|} = \{i_1, \ldots, i_{|R|}\}$ retrieved as answer to q, R-Precision is defined as

$$R - Precision = \frac{|R \cap I_{|R|}|}{|I_{|R|}|}$$

and measures which fraction of the first $|R|$ retrieved items is actually relevant to q.

Definition 3.3.3. (**Recall**) Given a set of n items $I_n = \{i_1, \ldots, i_n\}$ that represent a query result, and the set of items R considered relevant to q, the recall is defined as

$$recall = \frac{|R \cap I_n|}{|R|}$$

and describes to which extent the items considered relevant have been retrieved.

Sometimes, for example in the area of question answering, it is good enough to know whether there is at least one relevant answer among the first n results. A simple measure that can be used for this purpose is answer-at-n.

Definition 3.3.4. (**Answer-at-n (a@n)**) Given a set of the first n items $I_n = \{i_1, \ldots, i_n\}$ of a query result, and the set of items R considered relevant to q, the answer-at-n value is defined as

$$a@n = \min(1, |I_n \cap R|).$$

Therefore, $a@n = 1$ if there is at least one relevant result among the first n results and 0 otherwise.

While the evaluation measures just described ignored the order of the ranked result lists, the evaluation measures we will describe in the following take the order of the ranked result lists into account. Thus, the order of the retrieved results has an impact on the value of the measures.

Definition 3.3.5. (**AP (average precision)**) Given a ranked result list RL of retrieval results for a query q and the set of results $R = \{d'_1, \ldots, d'_x\}$ in RL considered relevant, the average precision (AP) is defined as

$$AP(q, RL) = \frac{1}{|R|} \sum_{d'_i \in R} P@rank(d'_i, RL),$$

where $rank(d'_i, RL)$ is the rank of d'_i in RL.

The NDCG measure (normalized discounted cumulative gain) has been proposed in [JK02] and supports non-binary relevance assessments: relevance assessments have more than two relevance levels, such as non-relevant, marginally relevant, relevant, and highly relevant. Each relevance level is mapped to a relevance value which is a number such as highly relevant→3, relevant→2, marginally relevant→1, and non-relevant→0. As the original work computes the NDCG value in an algorithmic way and we want to keep the definition compact, we adapt the variant presented in [MRS08] to our notation.

Definition 3.3.6. (**NDCG at rank k**) Let $rel(q, d)$ be the relevance value attributed to document d for query q, RL the ranked list for query q, and RL_k the k^{th} result in RL. The NDCG value for q at rank k is defined as

$$NDCG(q, k) = Z_q \sum_{r=1}^{k} \frac{2^{rel(q, RL_r)} - 1}{\log(1 + r)},$$

where Z_q is a normalization factor such that a perfect ranked result list's NDCG at rank k is 1.

The RR (reciprocal rank) measure quantifies when the first relevant document is encountered in a result list.

Definition 3.3.7. (**RR (reciprocal rank)**) Let RL_q be a ranked list, and R_q the set of relevant items for query q. Then the reciprocal rank (RR) for q is defined as

$$RR(q) = \frac{1}{minrank(RL_q \cap R_q)},$$

where $minrank(RL_q \cap R_q)$ is the minimum rank of a relevant document in RL_q. If $RL_q \cap R_q = \emptyset$ (i.e., no relevant results are retrieved), $\frac{1}{minrank(RL_q \cap R_q)}$ is set to 0.

Common measures that are based on mean values for a set of topics (query load) are MAP (mean average precision) and MRR (mean reciprocal rank). While MAP averages over AP values, MRR averages over RR values:

Definition 3.3.8. (**MAP (mean average precision)**) Given a query load $Q = \{q_1, \ldots, q_m\}$ and a ranked result list of retrieved results for each query in Q, RL_j being the ranked list for query q_j and R_j the set of relevant items for query q_j. Then the mean average precision (MAP) for Q is defined as

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{m} AP(q_j, RL_j).$$

The MRR (mean reciprocal rank) averages over the reciprocal ranks of the first relevant retrieved result for each query in a query load.

Definition 3.3.9. (**MRR (mean reciprocal rank)**) Given a query load $Q = \{q_1, \ldots, q_m\}$ and a ranked result list of retrieved results for each query in Q, RL_j being the ranked list for query q_j and R_j the set of relevant items for query q_j. Then the mean reciprocal rank (MRR) for Q is defined as

$$MRR(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{minrank(RL_j \cap R_j)},$$

where $minrank(RL_j \cap R_j)$ is the minimum rank of a relevant document in RL_j. If $RL_j \cap R_j = \emptyset$ (i.e., no relevant results are retrieved), $\frac{1}{minrank(RL_j \cap R_j)}$ is set to 0.

3.3.2 Measures for XML Retrieval

In [KPK+07] Kamps et al. describe the official retrieval effectiveness measures used for the Ad Hoc Track at INEX 2007. While in earlier years only XML elements were allowed for retrieval, INEX 2007 allowed arbitrary document parts, i.e., XML elements and passages. The Focused Task requires a ranked list of non-overlapping document parts (i.e., there is no document part in the ranked list which is enclosed or partially overlaps with any other document part from the ranked list). Submitting organizations are asked to provide for each query q ranked lists of 1,500 non-overlapping document parts L_q that are supposed to be most focused and relevant.

The amount of relevant information retrieved is measured in terms of the length of relevant text retrieved. Since 2005, INEX uses highlighting to get relevance assessments for the topics. Therefore, the evaluation is based on the number of relevant highlighted characters, not documents.

p_r is the document part assigned to rank r in the ranked list L_q of document parts returned by a retrieval system for a topic q[1]. $size(p_r)$ is the total number of characters contained in p_r and $rsize(p_r)$ the total number of characters in the highlighted relevant text part of p_r. $Trel(q)$ denotes the total number of characters in all highlighted relevant text for q.

The precision at rank r is defined as

$$P[r] = \frac{\sum_{i=1}^{r} rsize(p_i)}{\sum_{i=1}^{r} size(p_i)}$$

and measures which portion of the retrieved characters is relevant.

The recall at rank r is defined as

$$R[r] = \frac{\sum_{i=1}^{r} rsize(p_i)}{Trel(q)}$$

[1] The topic may be notationally omitted if it is clear from the context which topic we are considering.

and measures to which extent the retrieved characters cover characters from text considered relevant. Both precision and recall are similar to the earlier definitions but use characters instead of documents as evaluation units.

The interpolated precision at recall level x for query q is defined as

$$iP[x](q) = \begin{cases} max\{P[r] : R[r] \geq x\} & : \text{ if } x \leq R[|L_q|] \\ 0 & : \text{ else} \end{cases},$$

where $R[|L_q|]$ is the maximum recall over all retrieved documents L_q. It considers the maximum achievable precision after the returned results have achieved at least recall level x. If the recall level x exceeds the maximum recall level for L_q, the interpolated precision drops to 0.

The average interpolated precision measure for query q

$$AiP(q) = \frac{1}{101} \cdot \sum_{x \in SRL} iP[x](q)$$

builds the average over the interpolated precision at the 101 standard recall levels $SRL = \{0.00, 0.01, \ldots, 1.00\}$.

The mean average interpolated precision measure is defined as

$$MAiP = \frac{1}{\#T} \cdot \sum_{q \, \in \, T} AiP(q)$$

and expresses the performance across a set of topics T.

Chapter 4

Evaluation for Selected Score Models

In the first part of this chapter (Section 4.1) we present a lot of insightful experimental results from the original papers. However, they usually compare only a few of the proximity score models surveyed in Chapter 2. The second part of this chapter (Section 4.2) seeks to close this gap by performing a comparative analysis of a significant set of proximity score models in a single evaluation framework with four test beds.

4.1 Results from the Original Papers

In this section, we report the main results from the original papers and describe their experimental setups. When we describe the employed test beds, we first talk about the topics with the corresponding tracks/tasks and mention the employed document collection in brackets. More information about the test beds can be found in Chapter 3.

4.1.1 Linear Combinations of Scoring Models

Rasolofo and Savoy: Rasolofo and Savoy (cf. Section 2.4.1) use three Web Track Ad Hoc Task test beds from TREC-8 (TREC45-CR), TREC-9, and TREC-10 (both WT10g) with 125 multi-keyword queries without stopwords. They compare the retrieval quality of their proposed proximity-aware model to an Okapi BM25 baseline. Proximity scores help more with early (P@5) than with later precision. Average precision values for BM25 and the proposed model hardly differ for which the authors give two reasons: 1) Proximity scores only consider term pairs in a window size of five which limits the number of documents whose scores are influenced by proximity scores and 2) only the 100 documents with the highest BM25 scores are scored which may rule out potentially relevant documents beyond the top-100.

Sign tests at $p<0.05$ show that their approach significantly improves in AP over the baseline when evaluating over all queries from all three test beds. Considering single test beds, only for TREC-8, their approach significantly improves over the baseline.

Büttcher et al.: Büttcher et al. (cf. Section 2.4.2) perform two rounds of experiments: 1) They evaluate 100 topics from the TREC 2003 Robust Track (TREC45-CR) and 50 topics from the TREC 2004 Terabyte Track, Ad Hoc Task (GOV2). Like Rasolofo and Savoy, they compare their proximity-enhanced model to a BM25 baseline: a paired t-test shows significant improvements on GOV2 (P@10 at $p<0.02$, P@20 at $p<0.01$), but fails on TREC45-CR. 2) They split the GOV2 collection into 100 random chunks which are combined to form 10%, 20%, . . . , 90% of the GOV2 documents (20 subcollections per size). Their test bed uses the 100 queries from TREC Terabyte Track, Ad Hoc Tasks 2004 and 2005 (subcollections of GOV2). It turns out that the larger the document collection is, the more important the impact of term proximity scores gets for P@10 and P@20 values. The authors suspect, that, in large collections, it is more likely to accidentally find non-relevant documents that contain query terms; term proximity may help to find relevant ones. As the relative gain of proximity scores is higher for stemmed than for unstemmed queries, term proximity may help to find stem-equivalent terms that represent the same semantic concept. Average document length and effectivity of term proximity do not seem to be related.

Uematsu et al.: Uematsu et al. (cf. Section 2.4.3) use two test beds: 50 topics from the TREC-8 Web Track, Ad Hoc Task (TREC45-CR), and 30 IR Task Topics from IREX (IREX_IR collection). They compare precision and average query processing times when evaluating queries with document-, word-, and sentence-level indexes and report index sizes. A document-level index contains only (docid, tf(term,docid)) pairs, and word-level indexes contain additional term position information. The proposed sentence-level index contains for each term t a list of docids and the number of sentences where t occurs plus sentence positions. This is used to determine, for each document, the number of sentences with co-occurrences of all query terms per document. Indexes are compressed by means of dgap and v-byte encoding: due to smaller dgaps, sentence-level may compress better than word-level indexes. Sentence-level indexes lead to the highest early precision values for both test beds.

Document-level indexes are smallest, sentence-level indexes are as effective as word-level indexes, but smaller. Without positional information, document-level indexes are not as effective as the other indexes as they can only be used to compute BM25 scores without term proximity contributions. To index the TREC45-CR collection, the sentence-level index requires 900GB (39% larger than document-, 26% smaller than word-level index), for IREX_IR 210GB (25% larger than document-, 12% smaller than word-level index).

For the TREC-8 test bed, document- and sentence-level indexes' query processing times are comparable and a bit faster than the word-level indexes'. For the IREX_IR test bed, index granularities hardly influence the query processing speed.

Monz: Monz (cf. Section 2.4.4) uses test beds from the question answering (QA) Tracks of TREC-9, TREC-10 (both TIPSTER+TREC45), and TREC-11 (AQUAINT) to evaluate his *minimum span weighting (msw)* approach against the baseline *Lnu.ltc*.

He compares the percentages of questions that have at least one relevant document among the top-n results ($a@n$), and shows that msw outperforms the baseline on all test beds, especially for low n. The precision values decline for the TREC-11 test bed as it contains more difficult questions and the average number of relevant documents is lower than for the TREC-9 and TREC-10 test beds. Percentually, the precision values $P@n$ show higher gains than for $a@n$ at all cutoffs. For both metrics, with every test bed and at all cutoffs, the performance of msw significantly outperforms the baseline at $p<0.01$. The author fails to show a correlation between query length and average precision.

Tao and Zhai: Tao and Zhai (cf. Section 2.4.5) employ five TREC test beds, namely the 50 TREC-1 Ad Hoc Topics (AP, FR), DOE queries (DOE), and 50 TREC-8 Web Track, Ad Hoc and Small Web Topics (TREC45-CR, WEB2g). The authors report average values of five individual proximity distance functions (considered in isolation) for relevant and non-relevant documents. Ideally non-relevant documents should have higher distance values than relevant ones. It turns out that global measures (*Span* and $MinCover$) need a normalization as relevant documents tend to contain more query terms which span wider than in non-relevant documents. Local measures ($MinDist$, *AvgDist*, and $MaxDist$) perform better than global measures; $MinDist$ is likely to be the best proximity distance function on every test bed. Except for FR (maybe too few queries (21) are applicable which tends to support the null hypothesis), MAP values improve significantly (Wilcoxon signed rank test, $p < 0.05$) for $R_1 + MinDist$ and $R_2 + MinDist$ over the baselines KL-divergence and BM25, respectively. Early precision values are better for $R_1 + MinDist$ and $R_2 + MinDist$ than for the baselines. $R_1 + MinDist$ provides similar MAP values as the MRF approach used by Metzler and Croft [MC05] (cf. Section 2.7.3). Parameter sensitivity studies show that global proximity measures are less stable and accurate than local ones. Simple addition of KL-divergence and proximity measures cannot improve retrieval quality over KL-divergence.

4.1.2 Integrated Score Models

De Kretser and Moffat: De Kretser and Moffat (cf. Section 2.5.1) compare their locality to traditional document retrieval models using the AP metric for k=1,000 documents. They evaluate 150 long TREC-1-3 (TIPSTER Disks 1+2) and 49 short TREC-4 Ad Hoc Topics (TIPSTER Disks 2+3) on subcollections from TIPSTER Disk 2 (AP88, FR88, WSJ90-92, ZIFF89-90), newspaper AP88+WSJ90-92, and non-newspaper FR88+ZIFF89-90 parts.

As *document retrieval baselines*, they use 1) the standard cosine measure (`baseline1`) and 2) an approach which uses tf, qtf, pivoted document length normalization, and *idf*-normalization by maximum frequency which achieves the best overall performance in [ZM98] (`baseline2`). For *locality retrieval models*, the authors test four kernel shapes with damped and non-damped height.

For the short topics (7 distinct terms on average), with a few exceptions, their locality-based retrieval models improve over `baseline1` and the arc-shaped, damped height kernel outperforms `baseline2` on 5 of 6 document collections. For the long topics (43 distinct terms on average in description field), locality methods do not pay off except for FR88 which contains long documents where users save most time when pointed to passages by locality-based retrieval models.

Song et al.: Song et al. (cf. Section 2.5.2) compare early and average precision of their retrieval model (*newTP*) to BM25 and Rasolofo and Savoy's approach (*OkaTP*).

To this end, they use the 50 topics from the TREC-10 Web Track, Ad Hoc Task (WT10g) and 50 topics from the TREC-11 Web Track, Topic Distillation Task (.GOV) as test beds. Song et al. tune first on BM25, then the newTP parameters using the TREC-9 Web Track, Ad Hoc Task (WT10g) test bed: newTP's term proximity scores use much larger text windows (size=45) than OkaTP (size=5).

newTP significantly (paired t-test, p<0.05) outperforms BM25 in terms of P@5 and P@10 for both testbeds. For P@5, OkaTP outperforms newTP which indicates that OkaTP brings more documents with very close term pair occurrences to the top-5. For P@10, Song et al.'s approach outperforms OkaTP and BM25 which indicates that newTP can handle more distant term pairs better than OkaTP. newTP and OkaTP provide similar average precision values that both outperform BM25.

Mishne and de Rijke: Mishne and de Rijke (cf. Section 2.5.3) evaluate the impact of phrase and proximity terms and document structure on retrieval quality with two test beds: 50 queries from the TREC-12 and 75 queries from the TREC-13 Web Track, Topic Distillation Task (.GOV collection). They compare five approaches which use 1) single query terms from each topic as terms, no document structure (`baseline`), 2) all term-level n-grams from a topic as phrase terms (`phrases`), 3) phrase terms with weights proportional to term phrase frequencies in different fields (`phrases-b`), 4) all term-level n-grams from a topic as proximity terms with fixed distance length (`proximity`), and 5) with variable distance length (`prox-v`). The authors claim that using a multiple field representation for each document, phrase and proximity terms can help effectiveness and confirm Mitra et al. [MBSC97] that, for single field representations, given a good basic ranking model, phrases yield little or no improvement.

Phrase and proximity terms often help to provide higher effectiveness the less restrictive the variant in use (i.e., `prox-v` often outperforms `proximity` and `proximity` often outperforms `phrases`). `phrases-b` provides more stable results than `phrases.` Short queries often form linguistic phrases and rather gain effectiveness by phrase and proximity terms than longer queries. Those tend to consist of non-related sets of terms and may consequently suffer from a topic drift. Effectiveness gains for short queries predominate - starting at a query length of four terms, the effectiveness drops.

4.1.3 Language Models with Proximity Components

Lv and Zhai: Lv and Zhai (cf. Section 2.6.1) use four TREC test beds to evaluate their positional language model (PLM) approach: 50 TREC-1 Ad Hoc Topics (AP88-89 and FR), and 50 TREC-8 Web Track, Ad Hoc and Small Web Topics (WT2g and TREC45-CR). For the *best position strategy (BPS)*, they compare the effectiveness of proximity-based kernels. The KL-divergence model (with Dirichlet prior smoothing) returns initial results re-ranked with PLMs for $25 \leq \sigma \leq 300$: $\sigma \geq 125$ does best, and Gaussian kernels are usually preferable. Lv and Zhai claim that the Gaussian kernel is superior since it is the only kernel under view whose propagated count drops slowly for small distances $|i-j|$ (dependent terms are not always adjacent in documents), fast for moderate distances (boundary of term's semantic scope reached), and again slowly for large distances (all terms are only loosely associated). For the *multi-position strategy* with a single spread σ, a Gaussian kernel (with Dirichlet prior smoothing) does not yield noticeable improvements over BPS (k=1) such that, for one single σ, BPS can be considered a robust method for document ranking. For the *multi-σ strategy*, PLMs (σ_{PLM} flexible) and document language model ($\sigma_{LM} = \infty$) are linearly combined using a coefficient γ. Interpolation helps PLMs to be more robust and effective: the authors claim that PLMs represent proximity well although document-level retrieval heuristics are better represented by document LMs. The PLM approach performs best for small σ_{PLM} values (e.g., 25 or 75). For collections with larger *avgdl* values (i.e., WT2g and FR), PLMs need more weight (i.e., a larger γ) since their document LMs tend to be noisier.

Zhao and Yun: Zhao and Yun (cf. Section 2.6.2) use four test beds: the title fields of the 50 TREC-5 Ad Hoc Topics (AP88 and WSJ90-92), the title fields of 50 TREC-3 Ad Hoc Topics (WSJ87-92), and the OHSUMED Topics (OHSUMED). The compared approaches are 1) the KL-divergence model, 2) the KL-divergence model linearly combined with a proximity score model as used in [TZ07], and 3) the proposed proximity integrated language model (ProxLM) with different term proximity centrality measures.

The authors compare the best achievable performance of the term proximity centrality measures. $Prox_{SumProx}$ performs similarly well as $Prox_{MinDist}$ and both outperform $Prox_{AveDist}$. 2) performs better than 1) on all test beds except for OHSUMED whose queries are verbose. 3) outperforms 1) and 2) in terms of precision and MAP and can handle verbose queries very well: for OHSUMED, 3) always significantly (Wilcoxon signed rank test, p<0.05) outperforms 1) and 2).

The authors study how robust the approaches are if stopwords in queries are considered. To this end, they use the 23 Ad Hoc Topics from TREC-5 that contain at least one stopword on the collections AP88 and WSJ90-92. While, in the presence of stopwords, 1) is robust in terms of effectiveness, 2) fails for both collections. $Prox_{SumProx}$ used as centrality measure in 3) improves over 1), is robust to stopword occurrences, and superior to $Prox_{MinDist}$. The authors claim that stopwords occur frequently in documents, i.e., they are likely to occur close to other query terms which may highly influence proximity centrality scores for $Prox_{MinDist}$.

4.1.4 Learning to Rank

Metzler and Croft: Metzler and Croft (cf. Section 2.7.3) evaluate with four TREC test beds: the 150 Ad Hoc Topics from TREC-1 to TREC-3 (WSJ87-92 and AP88-90, respectively; both part of TIPSTER), 100 topics from the Web Track, Ad Hoc Task of TREC-9+10 (WT10g), and 50 topics from the Terabyte Track, Ad Hoc Task 2004 (GOV2). Documents are stemmed and stopwords removed during evaluation. The full independence (FI) setting (only cliques in T) serves as a baseline for the parameter-tuned, weighted MRF models. For the sequential dependence (SD) setting (cliques in T, U, and O), the authors evaluate MAP values for window sizes of 2, 8, 50, and ∞ and tune parameters separately for each window size. It seems that the window size only matters for the GOV2 collection: a size of 8 (which corresponds to the average length of English sentences) performs best and outperforms ∞-sized windows.

Hill climbing is used for parameter tuning which starts off with the FI setting (λ_T=1, λ_O=λ_U=0). The authors find that SD and FD significantly (paired t-test, p<0.05) improve MAP values for all testbeds compared to FI variants.

Svore et al.: Svore et al. (cf. Section 2.7.2) study the effect of different feature sets on effectiveness in web retrieval using stemmed English queries (up to 10 terms) sampled from a commercial search engine's query log. Each query has on average 150–200 assigned documents with 5-level relevance assessments. While the training set consists of 27,959 queries (of which 20% are used for validation), the test set consists of 11,857 queries. The authors compare the impact of phrases and proximity terms on early NDCG values and compare ten different models: 1) BM25, 2) train LambdaRank over BM25 features (λ**BM25**), 3) Rasolofo and Savoy's approach, 4) a bigram-version of 3), 5) Song et al.'s approach, 6) λ**BM25** with bigram features, 7) 6) with additional espan-based rc value from 5), 8) an approach that uses all espan goodness features and model feature sets (**Espan**), 9) **Espan** without formatting features, and 10) **Espan** without 3rd-party phrase features.

Espan significantly (t-test, p<0.05) outperforms all other models: phrase features and -even more- formatting features are important for retrieval effectiveness. To consider query characteristics, the queries in the test set are split by 1) length and 2) popularity. For popular queries, **Espan** significantly outperforms all other models. For short queries, removing phrase span features has only a small impact. In a full ranking model, **Espan** outperforms the other models. The experiments are not repeatable as neither queries nor assessments are disclosed.

Cummins and O'Riordan: Cummins and O'Riordan (cf. Section 2.7.4) make use of the LA, FBIS, and FR collections from TREC Disks 4 and 5 as test data. For each collection, the corresponding topic set is evaluated in two variants with stemming and stopword removal: 1) short (title field) and 2) medium length queries (title plus description fields). Furthermore, 63 topics are evaluated with the OHSUMED collection (title plus description fields only).

For each term-term proximity measure, for short and medium length queries separately, the average values for relevant and non-relevant documents are computed to see correlations. While average values for *min_dist, avg_min_dist* and *avg_match_dist* seem to be inversely correlated with relevance (i.e., larger average values for less relevant documents), *qt, sumtf,* and *prodtf* values seem to be directly correlated with relevance (i.e., larger average values for relevant documents).

Genetic programming is used to find a combination of a subset of the 12 proposed term-term proximity measures that form a learned proximity score. An FT subset of 69,500 documents from TREC Disk 4 and 55 topics (subset of Ad Hoc Topics from TREC-6+7) are used as training data.

$score_{ES}$ and a $score_{BM25}$ are used as baselines and linearly combined with the learned proximity score: on the training data, for ES (MAP as fitness metric), the proximity score generates significant improvements for $prox_5$ and $prox_6$ (Wilcoxon signed rank test, $p<0.05$), for BM25, proximity scores do not significantly improve the MAP values. An additional proximity-enhanced baseline is $score_{ES}$ combined with $MinDist$ as proximity function as used by Tao and Zhai which is not significantly better than $score_{ES}$ on the training data. For most test collections, $prox_6$ linearly combined with $score_{ES}$ also significantly improves over the $score_{ES}$ baseline, $prox_5$ still significantly improves for FBIS.

4.2 Comparative Analysis for Selected Score Models

In Section 4.1, we have seen that the original papers present a wealth of insightful experimental results. However, they usually only compare the effectiveness of just a few of the proximity score models described in Chapter 2. Therefore, it is difficult to assess which of the scoring models provides the best retrieval quality. We seek to close this gap by performing a comparative analysis of a significant set of proximity score models in one single evaluation framework with four test beds.

In our experiments, we use an open-source implementation of the MapReduce framework, Hadoop in version 0.20 on Linux. Hadoop runs on a cluster of 10 servers in the same network, where each server has 8 CPU cores plus 8 virtual cores through hyper-threading, 32GB of memory, and four local hard drives of 1TB each. The implementation has been done completely in Java 1.6.

We evaluate the retrieval quality for various parameter combinations and a set of individual scoring models. The evaluation is accelerated and partially only enabled by the distributed evaluation in the Hadoop framework. For each scoring model, we have implemented a separate class which can be fully customized and plugged into our evaluation driver. Each scoring model class includes a list of all parameters that are evaluated.

There is one file per test bed which contains all information that is necessary to perform the evaluation. This encompasses the document corpus, its characteristics (such as $avgdl$, N, and dt), the topic sets to be evaluated, and the relevance assessments. In addition, the file specifies query readers and document parsers plus optional Hadoop-related parameters to be used during evaluation; we employ the Galago Parser to parse the document collection.

The evaluation makes use of two jobs: in the map phase of the first job, we evaluate the query load for all documents and for each configuration. (A configuration consists of one scoring model with one parameter combination.) In the reduce phase of the first job, we aggregate, for each configuration and topic, retrieval quality statistics. The second job reconciles the per-topic, per-method, and per-metric results into averages per method and metric; the work is done exclusively in the reduce phase.

We will now briefly describe the various test beds we use across this section. For the Web Track, we evaluate the retrieval quality for the Web Tracks in 2000 and 2001 on the WT10g collection: topics 451-500 denote evaluations with Web Track Topics from TREC-9 (2000), and topics 501-550 denote evaluations with the Web Track, Ad Hoc Topics from TREC-10 (2001). For the Web Track, we additionally show the influence of limiting the evaluated topics to those that consist of at least two query terms, namely 451-500+2 and 501-550+2. This is intended to show the effects of proximity scores (which need at least two query terms to become effective). ALL encompasses all topics from both years (i.e., topics 451-550).

For the Robust Track, we evaluate the retrieval quality of the Ad Hoc Topics on TREC Disks 4 and 5 (without the Congressional Record data). While topics 301-350 denote evaluations with the Ad Hoc Topics from TREC-6 (1997), topics 351-400 denote evaluations with the Ad Hoc Topics from TREC-7 (1998). Topics 401-450 evaluate using the Web Track, Ad Hoc Topics from TREC-8 (1999), 601-650 represent the new topics from the TREC-12 Robust Track (2003), and 651-700 the new topics from the TREC-13 Robust Track (2004). ALL encompasses all topics from the five years, thereby considering the result quality values from topics 301-450 and 601-700 for each run.

For the Ad Hoc Tasks of the Terabyte Track, we evaluate the retrieval quality on the GOV2 collection. Topics 701-750, 751-800, and 801-850 denote evaluations with topics from TREC-13 (2004), TREC-14 (2005), and TREC-15 (2006), respectively. ALL encompasses all topics from the three years (topics 701-850).

For INEX, we evaluate the 68 Ad Hoc Track Topics from 2009 and the 52 Ad Hoc Track Topics from 2010 on the INEX Wikipedia collection from 2009. ALL encompasses the 120 topics from both years. In our evaluation, documents are considered relevant if they contain some characters marked as relevant.

4.2.1 Experimental Comparison of Scoring Models

For each test bed, we measure the retrieval quality using the NDCG@10, NDCG@100, P@10, P@100, and MAP retrieval metrics. Stop words are removed and query terms are stemmed. For each test bed, we compare the result quality of the scoring models by Büttcher et al., Rasolofo and Savoy, Zhao and Yun, Tao and Zhai, Lv and Zhai, Song et al., and de Kretser and Moffat. Furthermore, we evaluate LM with Dirichlet smoothing, ES, and BM25 as content-scores. We vary the parameters for each scoring model. For the Terabyte Track test bed, we can not evaluate Lv and Zhai's scoring model within a reasonable amount of time due to the collection size and positional language models that need to be constructed for every position in each document.

Evaluation Using Web Track Test Beds

Figures A.1 to A.3 in Appendix A show the best NDCG, precision, and MAP values for the Web Track test beds. Song et al.'s and Büttcher et al.'s scoring models have the highest NDCG values, Tao and Zhai's approach often performs similarly well. Tao and Zhai's scoring model and Büttcher et al.'s scoring model provide the highest precision values. Song et al.'s and Büttcher et al.'s scoring models perform best for the MAP metric. De Kretser and Moffat's approach performs worse than its competitors.

Restricting the test beds involving topics 451-500 and 501-550 to those with at least two query terms (451-500+2, 501-500+2) yields a higher overall retrieval quality, but does not influence the order of result quality among the scoring models.

Evaluation Using Robust Track Test Beds

Figures A.4 to A.6 in Appendix A show the best NDCG, precision, and MAP values for the Robust Track test beds. The best performing scoring models on the Robust Track test beds are the ones by Büttcher et al., Tao and Zhai, and Song et al.; there is no clear winner among these three models, usually they achieve similar retrieval quality values. Like for the other test beds, de Kretser and Moffat's approach falls behind the quality of the remaining scoring models.

Evaluation Using Terabyte Track Test Beds

Figures A.7 to A.9 in Appendix A show the best NDCG, precision, and MAP values for the Terabyte Track test beds. Büttcher et al.'s scoring model yields the highest retrieval quality for all test beds and retrieval metrics except for 751-800 with the MAP metric where it performs slightly weaker than Song et al.'s scoring model. For all other test beds Song et al.'s scoring model performs second best. BM25, Rasolofo and Savoy's scoring model perform similarly, but still good. For the NDCG and precision metrics, the Dirichlet smoothed language model and Zhao and Yun's scoring model yield similar retrieval quality, however often slightly weaker than BM25 and Rasolofo and Savoy's scoring model. For the MAP metric, Zhao and Yun's model falls behind the Dirichlet smoothed language model. De Kretser and Moffat's scoring model is far weaker than all other scoring models.

Evaluation Using INEX Test Beds

Figures A.10 to A.12 in Appendix A show the best NDCG, precision, and MAP values for the INEX test beds. Büttcher et al.'s scoring model yields the highest retrieval quality on all test beds (2009, 2010, and ALL) for all employed retrieval metrics. Song et al.'s approach always performs second best. Song et al.'s model outperforms BM25, Rasolofo and Savoy, LM with Dirichlet smoothing, and Zhao and Yun's approach; the latter four provide a similar retrieval quality and are a bit stronger than Lv and Zhai's approach. Tao and Zhai's scoring model and ES are similarly strong and a bit weaker than Lv and Zhai's scoring model. De Kretser and Moffat's approach is far behind.

4.2.2 Individual Scoring Models

This subsection details the parameter settings which have been evaluated for the scoring models. To keep the evaluation manageable, in this subsection, we restrict ourselves to evaluating all topics available for the test beds (i.e., topic set ALL). In the result tables we abbreviate the four resulting test beds by WEB, ROBUST, TERABYTE, and INEX.

BM25: For the (disjunctive) evaluation of the BM25 scoring model, we vary the parameters k_1 and b: $k_1 \in \{0.25, 0.4, 0.75, 1.0, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 2.0, 2.5\}$ (12 variants), $b \in \{0.25, 0.3, 0.5\}$ (3 variants), and 2 variants of *idf* (*idf*$_1$ and *idf*$_3$). k is always set to k_1. Therefore, in total $12 \cdot 3 \cdot 2 = 72$ runs are evaluated per test bed.

While for the MAP metric, smaller choices of b (0.25 and 0.3) work good for WEB and INEX, TERABYTE prefers larger b (0.3 and 0.5). For ROBUST, there does not seem to be a consistent best choice for b among the best parameter settings. k_1 should be small or medium-valued: $k_1 \leq 1$ yields the best results for WEB and ROBUST, $0.75 \leq k_1 \leq 1.2$ for TERABYTE, and $0.4 \leq k_1 \leq 1.0$ for INEX. The choice of the *idf* variant does not seem to be important for the retrieval quality. For the NDCG@10 metric, results tend to be better if k_1 is chosen larger, i.e., $k_1 \geq 1.5$ for WEB, $0.75 \leq k_1 \leq 1.2$ for ROBUST, $k_1 \geq 1.2$ for TERABYTE, and $0.75 \leq k_1 \leq 1.5$ for INEX. Good choices for b are similar as for MAP.

	optimize NDCG@10					optimize MAP				
Collection	k_1	k	b	*idf*	NDCG@10	k_1	k	b	*idf*	MAP
WEB	2.00	2.00	0.30	*idf*$_3$	0.3436	0.75	0.75	0.25	*idf*$_3$	0.2010
ROBUST	0.75	0.75	0.30	*idf*$_1$	0.4329	0.75	0.75	0.30	*idf*$_1$	0.2325
TERABYTE	1.60	1.60	0.50	*idf*$_1$	0.5004	1.00	1.00	0.30	*idf*$_1$	0.2973
INEX	1.00	1.00	0.30	*idf*$_1$	0.6148	0.75	0.75	0.30	*idf*$_1$	0.3389

Table 4.1: BM25: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.1 contains the optimal tuning parameter settings for BM25 with NDCG@10 and MAP values for all test beds.

Büttcher et al.: As described in Section 2.4.2, the proximity score part of Büttcher et al.'s scoring function is defined as

$$pscore(d, q) = \sum_{t \in q} min\{1, idf_1(t)\} \frac{acc_d(t) \cdot (k_1 + 1)}{acc_d(t) + K}.$$

We evaluate the effects of shrinking the influence of the *pscore* in Büttcher et al.'s scoring function, substituting $min\{1, idf_1(t)\}$ by $min\{minidf, idf_1(t)\}$, where $minidf \in \{0, 0.1, 0.2, \ldots, 1.0, 1.5, 2.0, 10000\}$ (14 variants). For the *cscore* part (i.e., a BM25 score variant), we evaluate the 72 BM25 tuning parameter combinations as described in the paragraph dealing with BM25. Thus, we evaluate $14 \cdot 72 = 1,008$ runs in total per test bed.

To achieve high MAP values, smaller values of b like 0.25 or 0.3 and relatively small values for k_1 are preferable on all test beds: on TERABYTE and INEX $0.4 \leq k_1 \leq 1.3$ perform best, on ROBUST $0.4 \leq k_1 \leq 1.0$, and on INEX $0.25 \leq k_1 \leq 1.0$. To achieve high NDCG@10 values, smaller values of b usually work well: b=0.25 performs best for WEB and INEX, and $b \in \{0.25, 0.3\}$ for ROBUST test beds. In contrast, TERABYTE achieves high NDCG@10 values for larger values of b, i.e., $b \in \{0.3, 0.5\}$. Medium-size choices of k_1 perform well: $0.75 \leq k_1 \leq 1.3$ for WEB, $0.4 \leq k_1 \leq 1.3$ for ROBUST, $0.6 \leq k_1 \leq 1.5$ for INEX test beds. Best runs for the TERABYTE test bed tend to have larger choices of k_1, i.e., $k_1 \geq 1.0$. Limiting the influence of the proximity score part makes sense, i.e., $minidf = 10000$ usually performs worse than values below 2: the best NDCG@10 runs use $0.5 \leq minidf \leq 1.0$ for WEB and ROBUST, $0.6 \leq minidf \leq 1.5$ for TERABYTE, and $0.6 \leq minidf \leq 2.0$ for INEX. The best MAP runs use $0.4 \leq minidf \leq 2.0$ for WEB, $0.5 \leq minidf \leq 1.5$ for ROBUST, $0.9 \leq minidf \leq 2.0$ for TERABYTE, and $0.7 \leq minidf \leq 2.0$ for INEX. The choice of the *idf*-variant has practically no impact on the result quality for all test beds and both retrieval metrics.

	optimize NDCG@10						optimize MAP					
Collection	k_1	k	b	*minidf*	*idf*	NDCG@10	k_1	k	b	*minidf*	*idf*	MAP
WEB	1.00	1.00	0.25	0.7	idf_1	0.3528	0.4	0.4	0.3	0.8	idf_3	0.2131
ROBUST	0.75	0.75	0.30	0.9	idf_1	0.4471	0.75	0.75	0.30	0.9	idf_1	0.2469
TERABYTE	1.60	1.60	0.30	1.0	idf_3	0.5199	0.75	0.75	0.30	1.5	idf_3	0.3257
INEX	1.30	1.30	0.25	1.0	idf_1	0.6398	0.75	0.75	0.25	2.0	idf_1	0.3764

Table 4.2: Büttcher et al.'s scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.2 contains the optimal tuning parameter settings for Büttcher et al.'s scoring model with NDCG@10 and MAP values for all test beds.

Rasolofo and Savoy: For the evaluation of Rasolofo and Savoy's scoring model, we use the same parameters as for BM25. In addition, $dist \in \{5, 10, 20, 10000\}$ (4 variants) is varied which specifies the text window width where pairs of query term occurrences influence each other's proximity contribution. Therefore, $72 \cdot 4 = 288$ parameter combinations are evaluated per test bed.

For WEB and INEX, smaller choices of b (0.25 or 0.3) usually generate better NDCG@10 and MAP values. For NDCG@10, both WEB and TERABYTE work best with medium and larger-valued k_1: $1.0 \leq k_1 \leq 2.5$. To yield good NDCG@10 values for ROBUST and INEX, k_1 should be chosen not that large: $0.4 \leq k_1 \leq 1.5$ and $0.75 \leq k_1 \leq 1.7$ perform best. To achieve good MAP performance, k_1 values should be chosen a bit smaller than for NDCG@10 values, i.e., $0.25 \leq k_1 \leq 1.2$ for ROBUST and INEX, $0.75 \leq k_1 \leq 1.4$ for TERABYTE, and $k_1 \leq 1.7$ for WEB.

There is a high impact of the choice of *dist*, especially for the NDCG@10 metric. Unfortunately, it is not clear whether to choose high or low *dist* values. The highest peaks are generated for the TERABYTE test bed: if chosen wrong, the NDCG@10 value can drop from 50% to 40%. Only the MAP value on WEB and ROBUST is not influenced much by the *dist* parameter. Furthermore, the choice of the *idf* version has a similar impact as *dist* and there is no tendency which *idf* version to prefer. Consequently, Rasofolo and Savoy's scoring model is difficult to tune.

	optimize NDCG@10						optimize MAP					
Collection	k_1	k	b	*dist*	*idf*	NDCG@10	k_1	k	b	*dist*	*idf*	MAP
WEB	2.00	2.00	0.30	10	idf_3	0.3436	0.40	0.40	0.25	5	idf_1	0.2059
ROBUST	0.75	0.75	0.50	10	idf_1	0.4329	0.40	0.40	0.30	20	idf_3	0.2276
TERABYTE	1.60	1.60	0.50	20	idf_1	0.5004	0.75	0.75	0.30	10,000	idf_3	0.2925
INEX	1.00	1.00	0.30	5	idf_1	0.6148	0.40	0.40	0.25	5	idf_1	0.3396

Table 4.3: Rasolofo and Savoy's scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.3 contains the optimal tuning parameter settings for Rasolofo and Savoy's scoring model with NDCG@10 and MAP values for all test beds.

Language Model with Dirichlet smoothing: For the Dirichlet smoothed language model we vary the smoothing parameter $\mu \in \{500, 750, 1000, 1250, 1500\}$ which leads to five evaluated runs per test bed.

The evaluation shows that the spread in result quality is usually very small so that the choice of the tuning parameter does not have a large influence. Nevertheless, often smaller choices of μ yield small improvements, e.g., for the MAP value on ROBUST and INEX as well as the NDCG@10 value on ROBUST and TERABYTE. For MAP on WEB and NDCG@10 on INEX, larger choices of μ are often slightly better. There is no clear tendency whether to choose μ small or large for MAP on TERABYTE and NDCG@10 on WEB.

	optimize NDCG@10		optimize MAP	
Collection	μ	NDCG@10	μ	MAP
WEB	750	0.3233	1,500	0.1988
ROBUST	500	0.4831	500	0.2340
TERABYTE	500	0.4255	750	0.3021
INEX	1,500	0.5984	500	0.3359

Table 4.4: Language Model with Dirichlet smoothing: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.4 contains the optimal tuning parameter settings for Language Model with

Dirichlet smoothing with NDCG@10 and MAP values for all test beds.

Zhao and Yun: For the evaluation of Zhao and Yun's score we use Dirichlet smoothing with smoothing parameter $\mu \in \{500, 1000, 2000, 3000, 5000\}$ (5 variants), scaling parameter $x \in \{1.5, 2.0, 2.5, 3.0, 4.0, 5.0\}$ (6 variants), $\lambda \in \{3.0, 5.0, 7.5, 10.0, 12.5\}$ (5 variants), and proximate centrality chosen between minimum distance, average distance, and summed distance (3 variants). This amounts to $5 \cdot 6 \cdot 5 \cdot 3 = 450$ evaluated parameter combinations per test bed.

To achieve high NDCG@10 values, usually $\mu = 500$ is a good choice on all test beds; only for INEX $\mu \in \{1000, 2000\}$ performs better. The best MAP-oriented runs use $\mu = 500$ for INEX and ROBUST, $\mu = 1000$ for TERABYTE, and $\mu = 2000$ for WEB test beds. For NDCG@10, while among the top runs for ROBUST and TERABYTE the average distance measure is most frequently used and for INEX the minimum distance measure is most frequently used as proximate centrality measure, there are no noticeable tendencies for WEB. For MAP, while the summed distance and minimum distance measures are most frequent among the top runs of ROBUST and INEX, there are no noticeable tendencies for WEB and TERABYTE. x and λ are very heterogeneously chosen among the top runs for all test beds; therefore, it is hard to give a general recommendation for choices of x and λ.

	optimize NDCG@10					optimize MAP				
Collection	μ	x	λ	$Prox_B(t_i)$	NDCG@10	μ	x	λ	$Prox_B(t_i)$	MAP
WEB	500	5	7.5	$Prox_{AvgDist}(t_i)$	0.3235	2000	5	5	$Prox_{MinDist}(t_i)$	0.1989
ROBUST	500	4	7.5	$Prox_{AvgDist}(t_i)$	0.4255	500	2.5	3	$Prox_{MinDist}(t_i)$	0.2340
TERABYTE	500	4	7.5	$Prox_{AvgDist}(t_i)$	0.4866	1000	4	3	$Prox_{SumDist}(t_i)$	0.2738
INEX	2000	1.5	10	$Prox_{AvgDist}(t_i)$	0.6008	500	2.5	5	$Prox_{SumDist}(t_i)$	0.3366

Table 4.5: Zhao and Yun's scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.5 contains the optimal tuning parameter settings for Zhao and Yun's scoring model with NDCG@10 and MAP values for all test beds.

Tao and Zhai: For the evaluation of Tao and Zhai's score we use a language model with Dirichlet smoothing and smoothing parameter $\mu \in \{100, 250, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000\}$ (11 variants), $\alpha \in \{0.1, 0.2, \ldots, 1.5, 2.0\}$ (16 variants), and the kernels MinDist, AvgDist, and MaxDist (3 variants). This results in $11 \cdot 16 \cdot 3 = 528$ evaluated runs per test bed.

The evaluation shows that the choice of the kernel has only a minor influence on result quality, although for the ROBUST and INEX test beds *MinDist* appears frequently among the best runs. Smaller and medium-valued choices of μ perform usually better than large choices. If one aims at optimizing MAP values, $\mu \in \{500, 1000, 1500\}$ works well for TERABYTE and INEX, $\mu \in \{250, 500, 1000\}$ for ROBUST, and $\mu \in \{1000, 1500, 2000, 2500\}$ for WEB. If one aims at optimizing NDCG@10 values, $\mu \in \{250, 500, 1000\}$ yields best values for TERABYTE ($\mu = 500$ works especially well), $\mu \in \{500, 1000, 1500, 2000\}$ for INEX, $\mu \in \{250, 500, 1000, 1500\}$ for ROBUST, and

$\mu \in \{500, 1000, 1500\}$ for WEB. Setting μ=500 works with all test beds and metrics. It is unclear how to select α: almost all values of α are represented within the best runs on all test beds with both metrics.

	optimize NDCG@10				optimize MAP			
Collection	μ	α	kernel	NDCG@10	μ	α	kernel	MAP
WEB	1,000	0.9	MaxDist	0.3269	2,000	0.8	MinDist	0.2057
ROBUST	500	1.1	MinDist	0.4207	500	0.7	MinDist	0.2313
TERABYTE	500	0.4	MaxDist	0.4517	1,000	0.6	MinDist	0.2572
INEX	500	0.8	MinDist	0.5205	500	1.0	MinDist	0.3080

Table 4.6: Tao and Zhai's scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.6 contains the optimal tuning parameter settings for Tao and Zhai's scoring model with NDCG@10 and MAP values for all test beds.

ES: As described in Section 2.2, Cummins and O'Riordan linearly combine the ES score with their proximity score combinations learned by Genetic Programming. As we do not have an implementation of this non-trivial Genetic Programming framework, we just use ES as another content-score without any parameters, generating only one run per test bed. Consequently, we cannot provide any optimal tuning parameter settings for NDCG@10 and MAP values.

Lv and Zhai: The scoring model proposed by Lv and Zhai builds one positional language model for each word position in the document. As a consequence, it is computationally very expensive so that we could not evaluate the TERABYTE test bed.

To further reduce computation costs, we have evaluated using the Gaussian kernel with Dirichlet prior smoothing as the Gaussian kernel is considered superior by Lv and Zhai (cf. Section 4.1.3). We have evaluated four parameter combinations, namely $\sigma \in \{25, 275\}$ and $\mu \in \{1000, 2000\}$. To rank, we scored each document by the best position in that document. For the three test beds (WEB, ROBUST, and INEX), we found that both for NDCG@10 and MAP used as retrieval quality metric, the combination $\mu = 1000$ and $\sigma = 275$ yielded the best results. Thus, the intercollection generalization results are perfect (always 1.0000) on this restricted set of parameter combinations for all pairs of test beds.

Song et al.: We evaluate Song et al.'s scoring model with $k_1 \in \{0.25, 0.4, 2.5\}$ (3 variants), $b \in \{0.3, 0.45\}$ (2 variants), two *idf* implementations, $x \in \{0, 0.25, 0.5, 0.55, 0.75, 1\}$ (6 variants), $y \in \{0, 0.25, 0.5, 0.75, 1\}$ (5 variants), and $d_{max} \in \{5, 45, 100\}$ (3 variants) which amounts to $3 \cdot 2 \cdot 2 \cdot 6 \cdot 5 \cdot 3 = 1,080$ parameter combinations.

Large k_1 (2.5) work best for NDCG@10 with the WEB and TERABYTE test beds. The best parameter combinations for INEX include medium and large choices of k_1 (i.e., 0.4 and 2.5), whereas we obtain the best values on ROBUST for small and medium choices of k_1 (0.25 and 0.4). For MAP, recommended choices of k_1 are more homogeneous: small and medium choices of k_1 (i.e., 0.25 or 0.4) yield the best MAP values for all test beds. For all test beds and both MAP and NDCG@10 metrics, $d_{max} = 5$ is very frequent among the top performing parameter settings. Except for TERABYTE and the NDCG@10 metric (where the choice of b is unclear), b set to 0.3 is common for the best runs. The choice of the *idf* version has only a minor impact on the result quality. Choices of x and y among the best runs are too heterogeneous to say anything meaningful about them.

	optimize NDCG@10								optimize MAP							
Collection	k_1	k	b	idf	x	y	d_{max}	NDCG@10	k_1	k	b	idf	x	y	d_{max}	MAP
WEB	2.5	2.5	0.30	idf_1	0.25	0.75	5	0.3471	0.4	0.4	0.3	idf_3	0.5	0.5	5	0.2114
ROBUST	0.4	0.4	0.3	idf_1	0.55	0.25	5	0.4418	0.4	0.4	0.3	idf_1	0.5	0.25	5	0.2440
TERABYTE	2.5	2.5	0.45	idf_1	0.25	0.25	5	0.5138	0.4	0.4	0.3	idf_3	0.75	0.25	5	0.3214
INEX	0.4	0.4	0.3	idf_1	0.5	0.0	5	0.6244	0.4	0.4	0.3	idf_1	0.55	0.25	5	0.3693

Table 4.7: Song et al.'s scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.7 contains the optimal tuning parameter settings for Song et al.'s scoring model with NDCG@10 and MAP values for all test beds.

De Kretser and Moffat: We evaluate de Kretser and Moffat's scoring model in conjunctive and disjunctive mode (2 variants), using the contribution functions triangle, cosine, circle, arc, circle', and arc' (6 variants), and the two algorithms to obtain a ranking for documents (2 variants) which generates $2 \cdot 6 \cdot 2 = 24$ runs in total.

De Kretser and Moffat's scoring model performs worse than all other scoring models we have evaluated. Nevertheless parameters have a high influence on result quality also for this scoring model; conjunctive evaluation of queries always provides higher retrieval quality than disjunctive evaluation which is especially high for TERABYTE and INEX test beds. For NDCG@10 advantages amount to about 12 and 17 percentage points, for MAP to around 7 and 5 percentage points, respectively. Only for the ROBUST test bed with MAP values, it is not clear whether one should use conjunctive or disjunctive query evaluation. To obtain a ranking for documents, choosing the first algorithm that greedily aggregates scores from different positions and documents (first algorithm) is often a bit better (below 2 percentage points) than considering the position with maximum score per document (second algorithm). This does not hold for the WEB test bed: if one wants to optimize NDCG@10 values, it remains unclear which ranking algorithm yields the better retrieval quality, for MAP values the second algorithm is often slightly better than the first algorithm.

	optimize NDCG@10				optimize MAP			
Collection	conj./disj.	kernel	algorithm	NDCG@10	conj./disj.	kernel	algorithm	MAP
WEB	conjunctive	circle'	2nd algorithm	0.2357	conjunctive	circle	2nd algorithm	0.1323
ROBUST	conjunctive	circle	1st algorithm	0.3340	disjunctive	circle'	1st algorithm	0.1493
TERABYTE	conjunctive	arc	1st algorithm	0.2370	conjunctive	circle	1st algorithm	0.1671
INEX	conjunctive	circle	1st algorithm	0.4092	conjunctive	circle'	1st algorithm	0.2005

Table 4.8: De Kretser and Moffat's scoring model: optimal tuning parameter setting with NDCG@10 and MAP values.

Table 4.8 contains the optimal tuning parameter settings for de Kretser and Moffat's scoring model with NDCG@10 and MAP values for all test beds. For all test beds, de Kretser and Moffat's approach falls behind the retrieval quality of the remaining scoring models.

4.2.3 Intercollection and Intracollection Generalization Results

In this subsection, we measure both intercollection and intracollection generalization performance of different scoring models.

To measure the *intercollection generalization* performance of a scoring model for a given evaluation metric, Metzler [Met06b] first computes the parameter combination for a training test bed that achieves the highest retrieval quality. This parameter combination is used as parameter combination for the test data (another test bed); the resulting retrieval quality m' is divided by the best retrieval quality m^* achievable on the test data to compute the effectiveness ratio $G = \frac{m'}{m^*}$.

We use one document collection with the corresponding topic set ALL as training test bed to check the retrieval quality for some parameter combinations. Then, we employ the parameter combination that yields the highest retrieval quality for the training data with a different document collection with the corresponding topic set ALL for that second collection (test data).

According to Metzler [Met06b], an ideal model that generalizes perfectly achieves an effectiveness ratio of 1. While effectiveness ratios below 0.90 indicate a scoring model's missing ability to generalize, the most reasonable retrieval models have an effectiveness ratio above 0.95. Table 4.9 shows the intercollection generalization results for various scoring models for both the NDCG@10 and the MAP metric.

Büttcher et al.'s scoring model and the LM approach with Dirichlet smoothing generalize especially well: all effectiveness ratios for both metrics are above 95%.

In most cases, for the NDCG@10 metric, the other scoring models do not generalize as well as these two approaches, but usually still generalize reasonably well: BM25's and Rasolofo and Savoy's effectiveness ratios are always above 93.63%; when trained on TERABYTE or INEX, the ratio even always exceeds 95%. Zhao and Yun's scoring model's effectiveness ratio is always above 93.13%; when trained on WEB, ROBUST or TERABYTE, the ratio overscores 95%. For Song et al.'s scoring model the effectiveness ratio overscores 93.18%; when trained on WEB, ROBUST or INEX, the ratio overscores 95%. Only de Kretser and Moffat's scoring model slightly underscores the 90% bound at a level of 87.91% when it is trained on WEB and tested with TERABYTE which may still be acceptable.

For the MAP metric, BM25 and Song et al.'s scoring model have an effectiveness ratio above 95% and thus generalize very well. Zhao and Yun's scoring model and Tao and Zhai's scoring model have a high effectiveness ratio of at least 94.33% and 94.28%, respectively; when trained on ROBUST, TERABYTE or INEX, the ratio always exceeds 95%. Rasolofo and Savoy's scoring model slightly underscores the 90% bound at a level of 88.17%; it may still be acceptable, especially when trained with ROBUST or TERABYTE. de Kretser and Moffat's scoring model is not able to generalize: when trained on ROBUST and tested on TERABYTE the effectiveness ratio is just 52.43%.

Scoring model		NDCG@10				MAP			
	Train\Test	WEB	ROBUST	TERABYTE	INEX	WEB	ROBUST	TERABYTE	INEX
BM25	WEB	-	0.9795	0.9498	0.9708	-	0.9956	0.9829	0.9895
	ROBUST	0.9363	-	0.9790	0.9613	0.9919	-	0.9989	1.0000
	TERABYTE	0.9708	0.9819	-	0.9512	0.9875	0.9887	-	0.9919
	INEX	0.9860	0.9974	0.9664	-	0.9919	1.000	0.9989	-
Büttcher et al.	WEB	-	0.9952	0.9775	0.9927	-	0.9947	0.9816	0.9727
	ROBUST	0.9848	-	0.9861	0.9795	0.9802	-	0.9890	0.9879
	TERABYTE	0.9789	0.9787	-	0.9696	0.9848	0.9906	-	0.9933
	INEX	0.9824	0.9917	0.9930	-	0.9907	0.9827	0.9979	-
Rasolofo, Savoy	WEB	-	0.9795	0.9498	0.9708	-	0.9767	0.8817	1.0000
	ROBUST	0.9363	-	0.9790	0.9613	0.9748	-	0.9633	0.9860
	TERABYTE	0.9708	0.9819	-	0.9512	0.9712	0.9993	-	0.9944
	INEX	0.9860	0.9974	0.9664	-	1.0000	0.9767	0.8817	-
LM, Dirichlet	WEB	-	0.9965	0.9928	0.9944	-	0.9650	0.9815	0.9673
	ROBUST	0.9940	-	1.0000	0.9951	0.9809	-	0.9824	1.0000
	TERABYTE	0.9940	1.0000	-	0.9951	0.9909	0.9913	-	0.9996
	INEX	0.9792	0.9844	0.9541	-	0.9809	1.0000	0.9824	-
Zhao, Yun	WEB	-	0.9997	0.9997	0.9875	-	0.9482	0.9534	0.9433
	ROBUST	0.9973	-	1.0000	0.9872	0.9745	-	0.9840	0.9999
	TERABYTE	0.9973	1.0000	-	0.9872	0.9831	0.9820	-	0.9882
	INEX	0.9492	0.9677	0.9313	-	0.9635	0.9986	0.9779	-
Tao, Zhai	WEB	-	0.9727	0.9717	0.9812	-	0.9526	0.9515	0.9428
	ROBUST	0.9782	-	0.9900	0.9971	0.9724	-	0.9916	0.9972
	TERABYTE	0.9745	0.9750	-	0.9632	0.9824	0.9805	-	0.9700
	INEX	0.9815	0.9917	0.9897	-	0.9666	0.9977	0.9855	-
Song et al.	WEB	-	0.9610	0.9836	0.9789	-	0.9982	0.9929	0.9944
	ROBUST	0.9670	-	0.9808	0.9907	0.9777	-	0.9889	0.9997
	TERABYTE	0.9724	0.9318	-	0.9413	0.9778	0.9869	-	0.9965
	INEX	0.9702	0.9984	0.9831	-	0.9775	0.9991	0.9924	-
De Kretser, Moffat	WEB	-	0.9416	0.8791	0.9244	-	0.9676	0.8924	0.9149
	ROBUST	0.9937	-	0.9980	1.0000	0.8070	-	0.5243	0.7134
	TERABYTE	0.9782	0.9992	-	0.9987	0.9608	0.9863	-	0.9856
	INEX	0.9937	1.0000	0.9980	-	0.9827	0.9813	0.9698	-

Table 4.9: Intercollection generalization results for various scoring models.

The *intracollection generalization* measure deals with how well a model trained on one topic set for a given collection generalizes to a different topic set on the same collection. Like for the intracollection generalization measure, the effectiveness ratio $G = \frac{m'}{m^*}$ is computed. To this end, the topic set is divided in two halves; one half is used for training, the other half for evaluation. This procedure is repeated 5,000 times to compute an average value for G which represents the intracollection generalization measure. All scoring models exhibit high intracollection generalization values between 98% and 100% on all test beds with both the MAP and NDCG@10 evaluation metrics. Therefore, we do not show the exact values.

4.2.4 Sensitivity Charts

Following Metzler's work [Met06a, Met06b], we compute entropy and spread values for the scoring models.

The *spread* of the effectiveness metric measures the quality difference between the parameter setting with the highest retrieval quality and the parameter setting with the lowest retrieval quality. Therefore, it gives an idea of how bad the results can get if we choose the wrong parameter values.

Given a topic set $\mathcal{Q}$ and the corresponding relevance assessments $\mathcal{R}$ with $\mathcal{T} = (\mathcal{Q}, \mathcal{R})$, the *entropy* is defined as

$$H = -\int_{\theta} P(\theta|\mathcal{T}) \log P(\theta|\mathcal{T}).$$

To estimate $P(\theta|\mathcal{T})$, the following procedure is performed B=5,000 times: in iteration b, we repeatedly sample a subset of topics from a test bed, i.e., if we have $|\mathcal{Q}|$ topics, we sample $|\mathcal{Q}|$ times with repetition. After sampling, we have obtained a subset $\mathcal{T}_b$ of $\mathcal{T}$, and determine the best parameter combination θ_b for $\mathcal{T}_b$.

After B iterations, Metzler estimates the posterior $P(\theta|\mathcal{T})$ s.t.

$$P(\theta|\mathcal{T}) = \frac{\sum_{i=1}^{B} \delta(\theta, \theta_i)}{B},$$

where $\delta(\theta, \theta_i)$ denotes Kronecker's delta. That means that one counts, for any given parameter combination θ, how often θ has been chosen as an optimal parameter combination during the B iterations and divides this number by the number of iterations B.

According to [Met06a], the spread and entropy provide a novel, robust way of looking at parameter sensitivity. Metzler claims that a model with high entropy and low spread is more stable than a model with low entropy but large spread; an ideal model features both low entropy and low spread. We think that this kind of evaluation is only fair when the number of evaluated parameter combinations is similar for all scoring models as the number of evaluated parameter combinations biases the results: the more parameter combinations of a scoring model are evaluated, the potentially higher its entropy and spread. The reason for different numbers of evaluated parameter combinations has two reasons: on the one hand, the number of parameters differs from scoring model to scoring model, on the other hand, evaluating many parameter combinations is infeasible for some scoring models as it is computationally too expensive such that experiments take arbitrarily long time.

Therefore, the experiments carried out for Lv and Zhai's scoring model (four settings), for the Dirichlet smoothed language model (five settings), and de Kretser and Moffat's scoring model (24 settings) are not directly comparable to the remaining scoring models. Anyway, we leave them in the sensitivity charts as the entropy and spread values can be considered as a lower bound for these scoring models: if more settings had been evaluated, the values would have potentially increased. In other words: if the spread or entropy values for the scoring model under consideration are already high with a small amount of evaluated settings, the scoring model would also perform bad or even worse given more evaluated settings.

When evaluating the sensitivity using NDCG@10, de Kretser and Moffat's scoring model usually features a comparably high spread. This is mainly due to the retrieval quality difference between runs using conjunctive and disjunctive evaluation. The Dirichlet smoothed language model just uses one parameter (μ) and therefore is less affected by high spreads. Lv and Zhai's model usually features the lowest entropy which is also caused by the low number of evaluated settings.

We show sensitivity charts in Appendix A to depict entropy and spread values for nine scoring models on the WEB, ROBUST, GOV, and INEX test beds.

Figures A.13(a) and A.13(b) show the sensitivity of nine scoring models on the WEB test bed for the MAP and NDCG@10 evaluation metric, respectively. Figures A.14(a) and A.14(b) show the sensitivity of scoring models on the ROBUST test bed for MAP and NDCG@10 evaluation metrics, respectively. Figures A.15(a) and A.15(b) show the sensitivity of scoring models on the TERABYTE test bed for MAP and NDCG@10 evaluation metrics, respectively. Figures A.16(a) and A.16(b) show the sensitivity of scoring models on the INEX test bed for MAP and NDCG@10 evaluation metrics, respectively.

For the scoring models with 24 or less evaluated parameter settings, the entropy value is naturally very low. Given that small amount of evaluated parameter settings, the spread of de Kretser and Moffat's scoring model is very high which renders it a scoring model which is difficult to tune.

Among the scoring models which had at least 72 evaluated parameter settings, Song et al.'s scoring model and Tao and Zhai's scoring model exhibit always the highest spread. BM25, Büttcher et al.'s, Zhao and Yun's, as well as Rasolofo and Savoy's scoring model have usually low spreads (except for Rasolofo and Savoy's model on TERABYTE with NDCG@10 where the spread is higher) and BM25 usually offers the lowest spread.

The entropy value of Büttcher et al.'s and Song et al.'s scoring model are usually highest. For the INEX test bed, Zhao and Yun's (both for MAP and NDCG@10) and Rasolofo and Savoy's scoring model (only for MAP) have a higher entropy. Furthermore, Zhao and Yun's approach has a higher entropy value than Büttcher et al.'s approach for the MAP value on TERABYTE.

In our setting, we think that the spread value is more meaningful than the entropy value as it measures how much retrieval quality can decrease if we choose the wrong parameter combination.

4.2.5 Summary

Büttcher et al.'s scoring model and LM Dirichlet smoothing provide the best intergeneralization values for both NDCG@10 and MAP. The other scoring models are slightly behind, but still exceed a level of 90% except de Kretser and Moffat's scoring model whose effectiveness ratio is just slightly above 50% for MAP. The intracollection generalization measures are excellent (98% to 100%) for all scoring models. Scoring models with low spread values include BM25, Büttcher et al.'s, and Zhao and Yun's scoring model.

With the exception of de Kretser and Moffat's scoring model, all surveyed proximity scoring models perform well in relevant sensitivity and generalization measures. We focus later on Büttcher et al.'s scoring model since it combines one of the best intercollection generalization values and a low spread.

Chapter 5

Extensions

5.1 Introduction

This chapter deals with extensions to the proximity score model proposed by Büttcher et al. [BCL06] described in Section 2.4.2 and provides an extensive experimental study to investigate their impact on retrieval quality.

Term proximity has been a common means to improve effectiveness for text retrieval, passage retrieval, and question answering, and several proximity scoring functions have been developed in recent years. Sections 2.4 to 2.7 survey a selection of proximity scoring models developed for text retrieval during the last decade. For XML retrieval, however, proximity scoring has not been similarly successful. To the best of our knowledge, there is only one single existing proposal for proximity-aware XML scoring. This proposal has been authored by Beigbeder and was initially described in [Bei07] and extended towards full boolean query support in [Bei10] by the same author. It computes, for each position in an element, a fuzzy score for the query, and then computes the overall score for the element by summing the scores of all positions and normalizing by the element's length. We provide a more detailed description of this scoring model in Section 5.2.3.

The contributions of this chapter are two-fold: 1) In Section 5.2 we propose one of the first XML score models that uses proximity information. This part is based on our work published in [BS08b] and [BST08] which presents a proximity score for content-only queries on XML data. We describe how to adapt the existing scoring model proposed by Büttcher et al. [BCL06] towards XML element retrieval by taking into account the document structure when computing the distance of term occurrences. 2) In Section 5.3, by means of a case study, we rigorously analyze the potential of explicit phrases for retrieval quality and compare it to the proximity score used in [SBH$^+$07]. This part is based on our work published in [BBS10].

5.2 XML

In this section, we introduce some XML-related background and describe Beigbeder's approach for proximity-enhanced XML retrieval [Bei07, Bei10] as well as his experimental results. Then, we present our own XML score model for content-only queries on XML data that uses proximity information published in [BS08b] and [BST08]. We show experimental results for two test beds and present a new evaluation metric.

5.2.1 XML Background

In the context of the INEX workshop, documents are Wikipedia articles that have been annotated with XML tags. For our experiments in this chapter, we use the Wikipedia collection used for INEX during the years 2006 to 2008 (cf. Section 3.2.2) which contains tags that can be classified into two categories [DG06a]: a) language-independent general tags that carry structural information derived from the Wikitext format, and b) language-dependent template tags which describe repetitive information. Examples for general tags include `article`, `section`, `p` (which stands for paragraph), `title`, various forms of links (e.g., `collectionlink` and `unknownlink`), and emphasis levels (e.g., `emph2` and `emph3`). Template tags always start with `template_` and vary depending on the language of the Wikipedia collection in use.

According to the W3C recommendation from 26 November 2008 (`http://www.w3.org/TR/xml/`), elements are either delimited by start tags and end tags (e.g., `<article>` and `</article>`), or, for empty elements, by an empty-element tag (e.g., `<br/>`). Each element has a type, identified by name (generic identifier (GI)), and may have a set of attribute specifications. Each XML document can be represented as an element tree.

Nodes represent elements and directed edges indicate parent-child relationships between elements in the document under consideration. If the complete collection including links is considered, the tree structure is converted into the more general graph structure, links being considered as directed edges which may generate loops. Hence, XML retrieval aims at retrieving subtrees/subgraphs from the collection graph as results to an issued query.

According to [KGT$^+$08], the two main research questions for the INEX Ad Hoc Track are 1) whether the document annotation helps to identify the relevant portion of a document, and 2) how focused retrieval compares to traditional document-level retrieval.

5.2.2 Notation

To discuss proximity scoring models for XML elements, we adapt the notation introduced for text retrieval in Section 2.1.2 to the XML element retrieval setting where term positions in an element are defined analogously to term positions in documents.

Definition 5.2.1. (**element length; position-related notation**) Given an element e in an XML document d, the *element length of* e is defined as $l_e = |e|$ and corresponds to the number of term occurrences in e. Given e with length l_e, we denote the *term occurring at position i of e* by $p_i(e)$, $1 \leq i \leq l_e$; if the element is clear from the context, we simply write p_i. For a term t, we capture the *positions in element e where t occurs* by $P_e(t) = \{i | p_i(e) = t\} \subseteq \{1, \ldots, l_e\}$; if e is clear from the context, we write $P(t)$. Given a query $q = \{t_1, \ldots, t_n\}$, we write $P_e(q) := \cup_{t_i \in q} P_e(t_i)$ for the *positions of all query terms in element e*, again omitting the suffix e if the element is clear from the context. Given a set of positions $P \subseteq \{1, \ldots, l_e\}$ and an element e, we write $T_e(P)$ to denote the *set of terms at the positions of $P \subseteq \{1, \ldots, l_e\}$ in e*. Precisely, $T_e(P) = \{t | \ i \in P \wedge p_i(e) = t\}$.

Definition 5.2.2. (**set of pairs of adjacent query term occurrences; set of pairs of all query term occurrences**) We denote *pairs of query terms that are adjacent to each other* (there might be non-query terms in between) *in an element e* by

$$Q_{adj,e}(q) := \{(i,j) \in P_e(q) \times P_e(q) \mid (i < j) \wedge \forall k \in \{i+1, \ldots, j-1\} : k \notin P_e(q)\}.$$

Pairs of query terms within a window of dist positions in an element e are defined as

$$Q_{all,e}(q, dist) := \{(i,j) \in P_e(q) \times P_e(q) \mid (i < j) \wedge (j - i \leq dist)\}.$$

Please note that in this case, the query terms need not to occur consecutively in e. $Q_{all,e}(q)$ is the same but employs a window size of l_e.

5.2.3 Related Work by Beigbeder

This section describes a proposal for proximity-aware XML scoring that has been authored by Beigbeder and was initially described in [Bei07] and extended towards full boolean query support in [Bei10] by the same author. He transfers a score model akin to the one proposed by de Kretser and Moffat for text retrieval in [dKM99] (cf. Section 2.5.1) to XML retrieval.

The approach answers boolean queries. To this end, it introduces several modes to combine the impacts of the contribution function at position x in document d:

- conjunctive mode: $c_{q_1 \wedge q_2}(x) = min(c_{q_1}(x), c_{q_2}(x))$,
- disjunctive mode: $c_{q_1 \vee q_2}(x) = max(c_{q_1}(x), c_{q_2}(x))$, and
- complement mode: $c_{\neg q_1}(x) = 1 - c_{q_1}(x)$,

where q_i is a boolean query.

If q_i is a term t, the value of the contribution function at position x in d is defined as $c_t(x) = max_{l \in P_d(t)} c'_t(x, l)$, where $c'_t(x, l) = max(0, 1 - \frac{|x-l|}{s})$. The contribution function c'_t is triangle-shaped, its height h_t is 1, and the spread s is considered a built-in parameter which is kept constant for all terms. For the most frequent elements in the Wikipedia collection used for INEX 2008, Beigbeder distinguishes between manually chosen title-like elements and section-like elements. While title-like elements encompass `name`, `title`, `template`, and `caption` elements, section-like elements consist of `article`, `section`, `body`, `figure`, `image`, `page`, and `div` elements.

To score full XML documents or passages in XML documents, query terms that occur in title-like elements can extend their influence to the full content of the element and recursively to the elements it contains. The intention of that so-called propagation mode is to reflect the descriptive property of the title element for the section element it entitles. Thus, given any positions l and x, if l is located in a title-like element e_t and x in a section-like element e_s that is entitled by e_t, it follows that $c_t(x) = 1$. For all positions x located in any title-like element, the contribution $c_t(x)$ is 1.

The score of a document or passage p, respectively that starts at position x_1 and ends at position x_2 is defined as

$$score(q,p) = \frac{\sum_{x_1 \le x \le x_2} c_q(x)}{|p|},$$

where the document/passage length $|p| = x_2 - x_1 + 1$ is used for score normalization.

The approach requires a mapping that keeps information whether a given term position belongs to a title-like or section-like element. For the propagation mode, additional descendant information for elements is necessary to decide where to propagate scores. Descendants information for XML documents can be kept in pre-/post-order trees, for example.

Experimental evaluation: For the experimental evaluation, Beigbeder evaluates the 70 assessed topics from the INEX 2008 Ad Hoc Track (with the INEX Wikipedia collection 2006–2008), drops importance modifiers, and mostly uses keywords in the title field. Some topics are modified before evaluation to fit the boolean model better (e.g., *spanish classical guitar players* is modified to *spanish (classical* | *classic) guitar players*).

Beigbeder varies the spread s and evaluates three approaches, retrieving only section-like elements: 1) NP-NS (no propagation, no structure) where the structure is ignored (term proximity only used as in text retrieval), 2) NP-S (no propagation, structure) where only section-like elements are retrieved, term proximity influence ends at the boundaries of section-like elements, terms in title-like elements are not propagated, and 3) P-S (propagation, structure) where title-element terms' propagation is enabled.

The best run's (P-S, s=10) iP-Value of 0.69490141 outperforms the best INEX 2008 run(0.68965708). For $s \in \{5, 50, 500\}$, Precision-Recall curves are highest for P-S whose iP value benefits from small choices of s.

5.2.4 Proximity Scoring for XML

This section presents our proximity-enhanced score model for XML element retrieval based on our work published in [BS08b] and [BST08] which answers content-only queries on XML data.

To compute a proximity score for an element e with respect to a query with multiple terms $q = \{t_1, \dots, t_n\}$, we first compute a linear representation of e's content that takes e's position in the document into account, and then apply a variant of the proximity

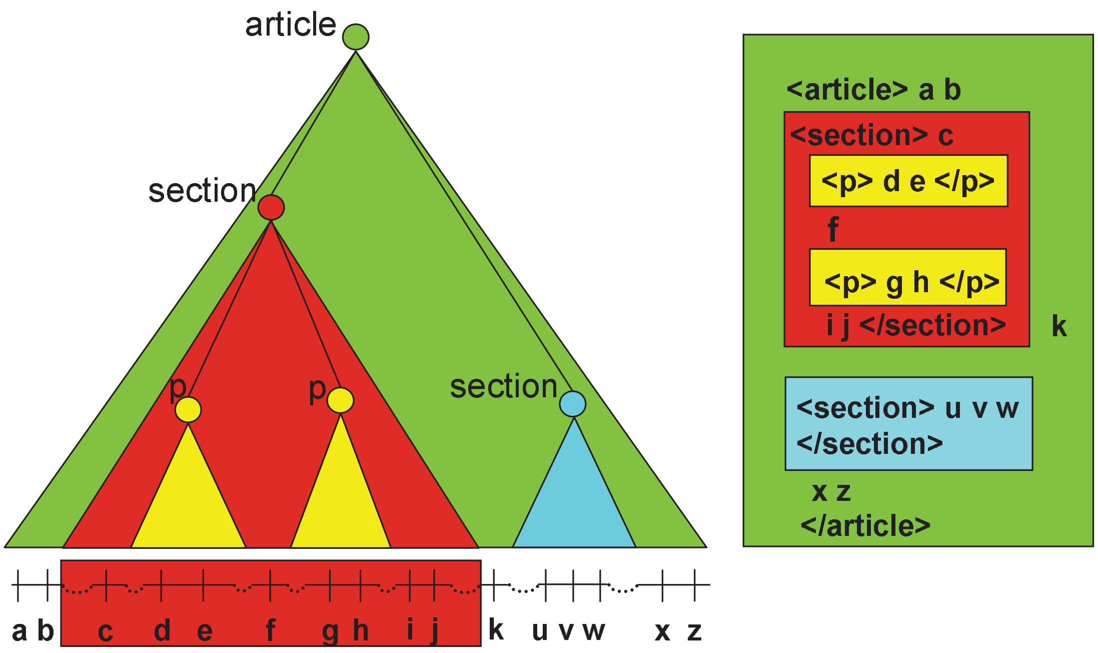


Figure 5.1: An XML document and its linearization.

score by Büttcher et al. [BCL06] on that linearization. This variant has been first proposed in [SBH+07] and will be described in detail in Section 7.2.2.

Figure 5.1 shows an example for the linearization process. We start with the sequence of terms in the element's content. Now, as different elements often discuss different topics or different aspects of a topic, we aim at giving a higher weight to terms that occur together in the same element than to terms occurring close together, but in different elements. To reflect this in the linearization, we introduce virtual gaps at the borders of certain elements whose sizes depend on the element's tag (or, more generally, on the tags of the path from the document's root to the element). In the example, gaps of `section` elements may be larger than those of `p` (paragraph) elements, because the content of two adjacent `p` elements within the same `section` element may be considered related, whereas the content of two adjacent `section` elements could be less related. Some elements (like those used purely for layout purposes such as `bold` or for navigational purposes such as `link`) may get a zero gap size. The best choice for gaps depends on the collection; gap sizes are chosen manually in our experiments.

Based on the linearization, we apply the proximity scoring model of Büttcher et al. [BCL06] (cf. Section 2.4.2) for each element in the collection to find the best matches for a query $q = \{t_1, \ldots, t_n\}$ with multiple terms.

To allow index precomputation without knowing the query load, we reuse the modified variant proposed in [SBH+07] (detailed explanations can be found in Section 7.2.2) that does not only consider pairs of adjacent query term occurrences in documents, but **all pairs** of query term occurrences (**not necessarily adjacent**). We further generalize the approach to score elements instead of documents, so the query-independent term weights in the formulas are not inverse *document* frequencies but inverse *element* frequencies

$$ief(t) = \log_2 \frac{N - ef(t) + 0.5}{ef(t) + 1},$$

where N is the number of elements in the collection and $ef(t)$ is the number of elements that contain the term t. Similarly, average and actual lengths are computed for elements.

Please note that, unlike [BSTW07], we do not use a tag-specific ief score

$$ief_A(t) = \log_2 \frac{N_A - ef_A(t) + 0.5}{ef_A(t) + 1},$$

where N_A is the tag frequency of tag A and $ef_A(t)$ is the element frequency of term t as to tag A, i.e., the number of elements (in documents of the corpus) with tag A that contain t in their full-content. We demonstrated in [BS08b] (and also in additional non-submitted results in [BSTW07]) that a global ief value for each term (i.e., $ief(t)$) achieves better result quality for content-only (CO) queries than tag-specific ief values (i.e., $ief_A(t)$).

The BM25 score of an element e for a query q is defined as

$$score_{\mathrm{BM25}}(e, q) = \sum_{t \in q} ief(t) \frac{tf(e,t) \cdot (k_1 + 1)}{tf(e,t) + K},$$

where $K{=}k{\cdot}[(1-b) + b \cdot \frac{l_e}{avgel}]$ with $avgel$ being the average element length in $\mathcal{C}$. b, k_1, and k are tuning parameters that are set to $b = 0.5$ and $k = k_1 = 1.2$, respectively. As for XML element retrieval the element length is important to keep up result quality, we do not ignore it in the proximity score component as in [SBH+07].

Hence, the proximity part of an element's score is computed by plugging the acc values into a BM25-style scoring function:

$$score_{prox}(e, q) = \sum_{t \in q} \min\{1, ief(t)\} \frac{acc(e,t) \cdot (k_1 + 1)}{acc(e,t) + K},$$

where

$$acc(e,t) = \sum_{\substack{(i,j) \in Q_{all,e}(q): \\ p_i = t, p_j = t', t \neq t'}} \frac{ief(t')}{(i-j)^2} + \sum_{\substack{(i,j) \in Q_{all,e}(q): \\ p_i = t', p_j = t, t \neq t'}} \frac{ief(t')}{(i-j)^2}$$

and K as well as the configurable parameters are set like for the BM25 score contribution.

The overall score is then the sum of the BM25 score and the proximity score:

$$score(e, q) = score_{\mathrm{BM25}}(e, q) + score_{prox}(e, q).$$

5.2.5 Experimental Evaluation

In order to evaluate our methods, in [BS08b] we used the standard INEX benchmark, namely the INEX Wikipedia collection [DG06a] with the content-only (CO) topics from the INEX Ad Hoc Task 2006. The 111 topics with relevance assessments are shown in Appendix C. Following the methodology of the INEX Focused Task, we computed, for each topic, a list of the 100 best non-overlapping elements with highest scores and evaluated them with the interpolated Precision metric used at INEX 2007[1] [KPK+07]. Details about the metric are given in Section 3.3.

When we check for significant improvements of an approach over the BM25 baseline, we first check for significance using the Wilcoxon signed rank test as it does not make any assumptions about the distribution of differences between pairs of results. If it fails at $p<0.10$, we try the paired t-test which assumes a normal distribution of differences between pairs of results. In all tables of this section, ‡ and † indicate statistical significance over the baseline according to the Wilcoxon signed rank test at $p<0.05$ and $p<0.10$, respectively. * and / indicate statistical significance over the baseline according to the paired t-test at $p<0.05$ and $p<0.10$, respectively.

Results for Document-Level Retrieval

For our first experiment, we evaluated how good our proximity-aware scoring is at determining documents with relevant content. We limited the elements in the result set to `article` elements, corresponding to complete Wikipedia articles, and considered different gap sizes, where we report

(1) gaps of size 0 for all elements,

(2) gaps of size 5 for `section` and 3 for `p` elements, and

(3) gaps of size 30 for `section` and `p` elements.

Approaches **(1)**-**(3)** all exploit proximity information in the form of $score_{prox}(e, q)$, and approaches **(2)** and **(3)** increase distances between query term occurrences in different elements by artificial gaps. We call the first approach gap-free, the latter two approaches gap-enhanced models. Additionally, we report results without proximity (i.e., only the BM25 score $score_{\mathrm{BM25}}(e, q)$ is used to rank elements) as baseline results. Our implementation first computed the 100 best results for the BM25 baseline and then additionally computed the different proximity scores for these results, re-ranking the result list.

Table 5.1 shows the results for document-level retrieval with stopword removal. If stemming is enabled, usage of the gap-free model that employs proximity information improves every iP and MAiP value compared to the baseline, and gaps help additionally (except for iP[0.01]). The same holds if stemming is disabled, this time without any

[1]Due to a bug reported for the original INEX implementation, we used a Java-based reimplementation of the metric.

	stemming, stopword removal				no stemming, stopword removal			
metric	baseline	(1)	(2)	(3)	baseline	(1)	(2)	(3)
iP[0.01]	0.6610	0.6916†	0.6912†	0.6859/	0.6721	0.7043‡	0.7045‡	0.7046‡
iP[0.05]	0.5630	0.5918*	0.5953*	0.5930*	0.5701	0.5904*	0.5954†	0.5953†
iP[0.10]	0.5339	0.5496/	0.5545*	0.5521/	0.5487	0.5644/	0.5685*	0.5684*
MAiP	0.2682	0.2795‡	0.2804‡	0.2798‡	0.2617	0.2717‡	0.2725‡	0.2730‡

Table 5.1: Results for document-level retrieval with stopword removal.

exception. In both cases, we get very significant improvements for proximity scores over the baseline. With only a few exceptions, gap-enhanced approaches can further improve the iP result quality over gap-free approaches. For the MAiP metric all approaches achieve significant improvements over the baseline with the Wilcoxon signed rank test at $p<0.05$.

	stemming, no stopword removal				no stemming, no stopword removal			
metric	baseline	(1)	(2)	(3)	baseline	(1)	(2)	(3)
iP[0.01]	0.6440	0.6233	0.6186	0.6163	0.6660	0.6068	0.6157	0.6187
iP[0.05]	0.5476	0.4986	0.4964	0.4984	0.5579	0.5081	0.5173	0.5209
iP[0.10]	0.5117	0.4620	0.4604	0.4596	0.5230	0.4775	0.4853	0.4888
MAiP	0.2543	0.2444	0.2432	0.2434	0.2487	0.2360	0.2369	0.2375

Table 5.2: Results for document-level retrieval without stopword removal.

Table 5.2 depicts the impact of missing stopword removal on the results for document-level retrieval. The results clearly demonstrate that stopword removal is crucial if we do not want to risk decreasing result quality with proximity scores compared to the baseline. If the query contains stopwords, the loss of result quality can be attributed to stopword occurrences near other query terms in some documents; as all pairs of query terms are considered if a document is to be scored, they generate an increased proximity contribution for the corresponding document. We think that these pairs are less meaningful (i.e., carry less semantics) than pairs of non-stopwords.

Gap-enhanced models cannot resolve the issue of losing result quality against the baseline. They just reduce the losses if stemming is disabled but do not get even close to the baseline's result quality. Consequently, all significance tests to show improvements over the baseline fail.

In summary, stopword removal is mandatory to get high retrieval quality for document-level retrieval and gap-enhanced approaches can often help additionally to improve the retrieval quality. In most cases, runs that are based on disabled stemming have slight advantages for the absolute iP values over those runs that use stemming. We get the best MAiP values when stemming is enabled and stopwords are removed.

Results for Element-Level Retrieval

We now evaluate the performance of proximity-aware scoring for element-level retrieval, where we limit the set of elements in the result list to those with `article`, `body`, `section`, `p`, `normallist`, and `item` tags for efficiency reasons; initial experiments with all tags yielded similar results. As we had to remove overlap, we first computed the best 200 elements for the BM25 baseline, for which we then computed the proximity scores, resorted the list according to the new scores, and removed the overlap between elements. Whenever two elements overlapped, we kept the element with the highest score.

	stemming, stopword removal				no stemming, stopword removal			
metric	baseline	(1)	(2)	(3)	baseline	(1)	(2)	(3)
iP[0.01]	0.6589	0.6847‡	0.6746†	0.6753†	0.6624	0.6659	0.6681	0.6677
iP[0.05]	0.5344	0.5591‡	0.5534‡	0.5544‡	0.5143	0.5274†	0.5280†	0.5263
iP[0.10]	0.4482	0.4680‡	0.4643‡	0.4643‡	0.4270	0.4441†	0.4413	0.4370
MAiP	0.1793	0.1870‡	0.1855‡	0.1854‡	0.1617	0.1670	0.1669†	0.1666†

Table 5.3: Results for element-level retrieval with stopword removal.

Table 5.3 illustrates the results for element-level retrieval with stopword removal. The best results and most significant improvements in element-level retrieval can be achieved if stemming is enabled. While the gap-free approach shows significant improvements over the baseline (Wilcoxon signed rank test at p<0.05, for every metric), the gap-enhanced approaches slightly lose absolute result quality compared to the gap-free approach. However, this does not overly harm the significance of improvements of gap-enhanced approaches over the baseline. For all metrics, except for iP[0.01], we achieve significant improvements with the Wilcoxon signed rank test at p<0.05, for iP[0.01] the improvements are still significant according to the Wilcoxon signed rank test, but only at p<0.10. If stemming is disabled, the usage of proximity improves every iP value compared to the baseline, but gaps help slightly only for early iP values. Significant improvements over the baseline using the Wilcoxon signed rank test at p<0.10 can be realized only for later iP values.

In general, compared to stemming with stopword removal, no stemming with stopword removal achieves less significant improvements for approaches **(1)**-**(3)** over the baseline (if at all) as well as a lower absolute result quality.

	stemming, no stopword removal				no stemming, no stopword removal			
metric	baseline	(1)	(2)	(3)	baseline	(1)	(2)	(3)
iP[0.01]	0.6409	0.6539*	0.6524/	0.6484	0.6079	0.6050	0.6045	0.6047
iP[0.05]	0.5077	0.5165/	0.5163/	0.5072	0.4736	0.4828	0.4764	0.4758
iP[0.10]	0.4042	0.4139	0.4136	0.4075	0.3627	0.3766‡	0.3700†	0.3702†
MAiP	0.1452	0.1535‡	0.1533‡	0.1504†	0.1267	0.1331‡	0.1308†	0.1307†

Table 5.4: Results for element-level retrieval without stopword removal.

Table 5.4 shows the results for element-level retrieval without stopword removal. Gap-free models improve every iP and MAiP value over the baseline, except for iP[0.01] if stemming is disabled. Gap-enhanced models slightly lose on absolute result quality compared to gap-free models but still frequently beat the baseline.

When we combine stemming with stopword removal, we achieve the best and most significant results for element-level retrieval. Gaps, however, help only for early iP values if stemming is disabled and stopwords are removed. Stopword removal is more important for document-level retrieval than for element-level retrieval if we want to obtain a good result quality, but our approaches benefit from stopword removal at both retrieval granularities. The structure-aware proximity score for XML retrieval that we have presented helps to improve the retrieval effectiveness of gap-free approaches for document-level retrieval, but does not show a similar effect for element-level retrieval. An automated selection of gap sizes by means of relevance feedback techniques could improve the result quality.

5.2.6 Additional Experiments for INEX 2008

This subsection describes additional experiments we have carried out for INEX 2008. It extends the experiments from Section 5.2.5 which used the 111 CO topics from the INEX Ad Hoc Task 2006 by another test bed (including the same document collection) used in the INEX 2008 Ad Hoc Track. An overview of the INEX 2008 Ad Hoc Track has been authored by Kamps et al. and has been published in [KGT$^+$08]. 70 topics have been assessed for the INEX 2008 Ad Hoc Track which are depicted in Appendix C, Table C.4.

The choice of runs we submitted to the Focused Task at INEX 2008 [BST08] was based on earlier results from SIGIR 2008 [BS08b]: as iP[0.01] is the metric that ranks the runs in INEX, we have chosen the setting that provides the highest retrieval quality at iP[0.01] from our previous experiments (detailed in Section 5.2.5), i.e., no stemming, but stopword removal in document-level retrieval. The Focused Task aims at returning a ranked list of elements or passages in a focused way, i.e., returned elements must not overlap. According to [KGT$^+$08], participants were allowed to submit up to three element result-type runs per task and three passage result-type runs each, for the Focused, Relevant in Context, and Best in Context Task in the Ad Hoc Track. As we have only evaluated element result-type runs for the Focused Task, we have only been allowed to submit three runs as described in the following:

- `TopX-CO-Baseline-articleOnly`: this run considers the non-stemmed terms in the title of a topic (including the terms in phrases, but not their sequence) except terms in negations and stopwords. We restricted the collection to the top-level `article` elements and computed the 1,500 articles with the highest $score_{\text{BM25}}$ value as described in Section 5.2.4. Note that this approach corresponds to standard document-level retrieval. This run is comparable to the baseline approach for document-level retrieval with stopword removal and disabled stemming used in Section 5.2.5.

- `TopX-CO-Proximity-articleOnly`: this run re-ranks the results of the baseline run coined `TopX-CO-Baseline-articleOnly` by adding the proximity score contribution $score_{prox}$ as described in Section 5.2.4. We use gaps of size 30 for

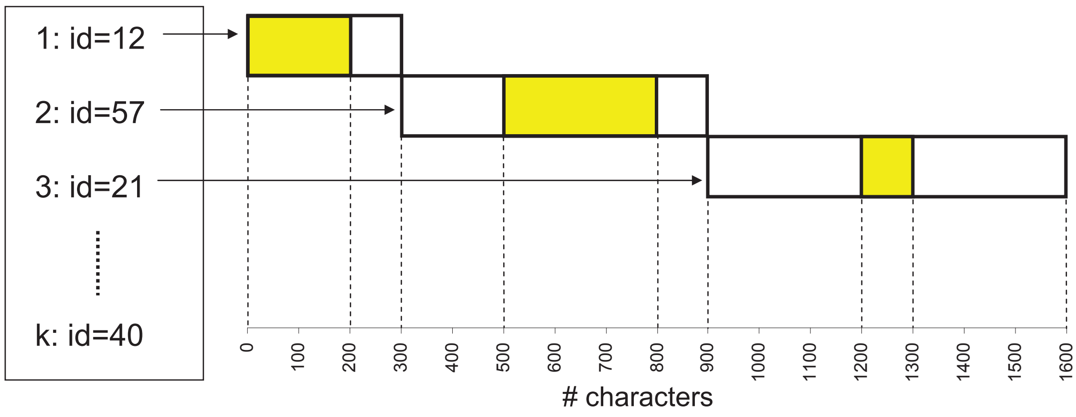


Figure 5.2: Example: illustration for metric P[#characters].

`section` and `p` elements. This run is comparable to the gap-enhanced approach **(3)** used in Section 5.2.5. Due to the limited number of submittable runs to INEX 2008, we could not evaluate different gap sizes.

- `TopX-CO-Focused-all`: this element-level run considers the terms in the title of a topic without phrases and negations, allowing all tags for results. Note that, unlike our contributions from earlier years (e.g., [BSTW07]), we do not use a tag-specific *ief* score, but a single global *ief* value per term. We demonstrated in [BS08b] that this achieves better result quality for CO queries than tag-specific *ief* values (cf. Section 5.2.4).

run/metric	iP[0.00]	iP[0.01]	iP[0.05]	iP[0.10]	MAiP
TopX-CO-Baseline-articleOnly	0.6700	0.6689	0.5940	0.5354	0.2951
TopX-CO-Proximity-articleOnly	0.6804	0.6795	0.5807	0.5265	0.2967
TopX-CO-Focused-all	0.7464	0.6441	0.5300	0.4675	0.1852

Table 5.5: Results: Focused Task INEX 2008, stopword removal, no stemming.

Table 5.5 shows the results for these runs. It is evident that element-level retrieval generally yields a higher early precision than document-level retrieval, but the quality quickly falls behind that of document-level retrieval which means that results become significantly worse than article-only runs starting at a recall level of 0.01.

This is reflected in the results: while the element-level run `TopX-CO-Focused-all` ranks at position 11 among 61 runs, the document-level runs rank at position 4 (`TopX-CO-Baseline-articleOnly`) and position 3 (`TopX-CO-Proximity-articleOnly`) among 61 runs, the last one being our best submitted run. Proximity scoring with gaps can in general help to improve early precision with document-level retrieval. MAiP values are almost equal for the document-level baseline `TopX-CO-Baseline-articleOnly` and the gap-enhanced model `TopX-CO-Proximity-articleOnly`.

Our experiments in SIGIR 2008 [BS08b] showed significant improvements of the gap-enhanced approach **(3)** over the baseline. Unfortunately, at INEX 2008 [BST08] comparable runs did not demonstrate equally significant improvements (significance levels are p=18.77% and 35.85% for paired t-test and Wilcoxon signed rank test, respectively).

As the iP metric returns the maximally achievable precision after the returned results have reached a recall level of at least x, this metric hides the points in the result sets where the retrieval quality originates from. Therefore, for analytical reasons, we have a look at the results using an alternative metric that measures the precision after x characters, abbreviated as P[x characters]. Figure 5.2 provides an example to illustrate how that alternative metric works. Assume that we want to calculate the precision value after 1,000 characters, P[1,000 characters]. We think of the result set as a characterwise concatenation of results for a given run; the evaluation measures the precision after reading the first 1,000 characters which corresponds to the retrieved number of characters carrying relevant content divided by the number of retrieved characters. Figure 5.2 characterwise aligns rectangles that represent the retrieved documents of a fictitious run, relevant characters are represented as yellow boxes. The first retrieved document of that fictitious run has id 12 and consists of 300 characters where the first 200 characters are considered relevant. The second retrieved document with id 57 consists of 600 characters of which 300 characters are relevant. As just the first 100 characters of the third document fit into the 1,000 characters limit, this document cannot generate a positive contribution to the precision (the relevant portion of this document starts after 300 characters only). Hence, the value for P[1,000 characters] is calculated as $\frac{200+300}{1{,}000} = 0.5$.

Figure 5.3 depicts the precision values after x characters for each of the three runs. It turns out that if we are interested in retrieving just a small amount of characters, it is worth considering to use the element-level run (leading up to 1,200 characters). Then the result quality of the element-level run deteriorates quickly and the proximity run outperforms the two other runs. Only very late, after 5,700 read characters, the baseline yields a slightly higher precision than the proximity run. Hence, to improve the retrieval quality, a hybrid approach could return the first 1,200 characters from the element-level run and fill the remaining characters with results from the proximity run.

5.3 Phrases

By means of a case study, we rigorously analyze the potential of explicit phrases for retrieval quality and compare it to the proximity score used in [SBH$^+$07]. This part is based on our work published in [BBS10].

5.3.1 Evaluating the Potential of Phrases

Phrases, i.e., query terms that should occur consecutively in a result document, are a widely used means to improve result quality in text retrieval [CCT97, CTL91, Fag87,

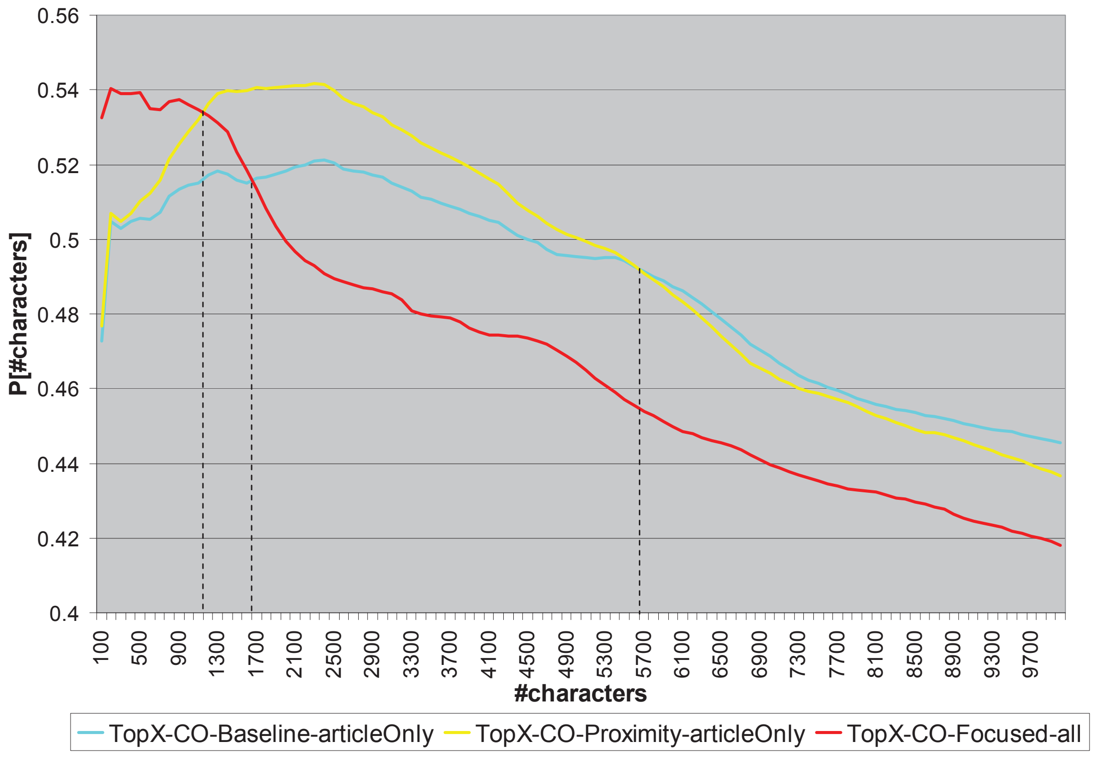


Figure 5.3: Comparison of the three runs: P[# characters] values.

LLYM04, MdR05], and a number of methods has been proposed to automatically identify useful phrases, for example [LLYM04, Z$^+$07]. However, there are studies indicating that phrases are not universally useful for improving results, but that the right choice of phrases is important. For example, Metzler et al. [MSC06] reported that phrase detection did not work for their experiments in the TREC Terabyte Track, and Mitra et al. [MBSC97] reported similar findings for experiments on news corpora.

The remainder of this chapter experimentally analyzes the potential of phrase queries for improving result quality through a case study on the TREC Terabyte benchmark. We study the performance improvement through user-identified and dictionary-based phrases over a term-only baseline and determine the best improvement that any phrase-based method can achieve, possibly including term permutations.

Experimental Setup

We did a large-scale study on the effectiveness of phrases for text retrieval with the TREC GOV2 collection, and the 150 topics from the TREC Terabyte Tracks 2004–2006 (topics 701–850) where we used the title only. More details about the collection and TREC can be found in Section 3.2.1. All documents were parsed with stopword removal and stemming enabled. We compared different retrieval methods:

- A standard BM25F scoring model [RZT04] as established baseline for content-based retrieval, with both conjunctive (i.e., all terms must occur in a document) and disjunctive (i.e., not all terms must occur in a document) query evaluation. The boosting weights are chosen as depicted in Table 5.6 and the same as the ones used in the GOV2 parser of the TopX search engine [TSW05].

- Phrases as additional post-filter on the results of the conjunctive BM25F, i.e., results that did not contain at least one instance of the stemmed phrase were removed. As the TREC topics do not contain explicit phrases, we considered the following ways to find phrases in the queries:
  - We performed a small user study where five users were independently asked to highlight any phrases in the titles of the TREC queries.
  - As example for a dictionary-based method for phrase detection, we matched the titles with the titles of Wikipedia articles (after stemming both), following an approach similar to the Wikipedia-based phrase recognition in [Z$^+$07].
  - To evaluate the full potential of phrases, we exhaustively evaluated the retrieval quality, i.e., precision for 10 results, of all possible phrases for each topic and chose the best-performing phrase(s) for each topic.
  - To evaluate the influence of term order, we additionally considered all possible phrases for all permutations of terms and chose the best-performing phrases, potentially after permutation of terms, for each topic.
- A state-of-the-art proximity score by Büttcher [BCL06] (described in Section 2.4.2) as an extension of BM25F, including the modifications from [SBH$^+$07]. This score outperformed other proximity-aware methods on TREC Terabyte; a thorough comparative experimental evaluation of various proximity-enhanced scoring models can be found in Section 4.2.

We additionally report the best reported results from the corresponding TREC Terabyte tracks, limited to title-only runs. When we checked for significant improvements over the baseline BM25F (conjunctive), we used both the Wilcoxon signed rank (WSR) test and the paired t-test.

Results

Our small user study showed that users frequently disagree on phrases in a query: on average, two users highlighted the same phrase only in 47% of the queries, with individual agreements between 38% and 64%. For each topic with more than one term, at least one user identified a phrase; for 43 topics, each user identified a phrase (but possibly different phrases). The same user rarely highlighted more than one phrase in a topic. Overall, our users identified 227 different phrases in the 150 topics.

Our experimental evaluation of query effectiveness focuses on early precision. We aim at validating if the earlier result by [MBSC97] (on news documents) that phrases do not significantly improve early precision is still valid when considering the Web.

tags	weight
`TITLE`	4
`H1, H2`	3
`H3-H6, STRONG, B, CAPTION, TH`	2
`A, META, EM, I, U, DL, OL, UL`	1.5

Table 5.6: Boosting weights BM25F.

topics	BM25F (conjunctive)	user 1	user 2	user 3	user 4	user 5
701-750 (TREC 2004)	0.536	0.512	0.534	0.504	0.546	0.536
751-800 (TREC 2005)	0.634	0.576	0.484	0.548	0.592	0.602
801-850 (TREC 2006)	0.528	0.518	0.500	0.514	0.546	0.526
average	0.566	0.535	0.506	0.522	0.561	0.554

Table 5.7: P@10 for user-identified phrases.

Table 5.7 shows precision values for the top-10 results when using the phrases identified by the different users (as strict post-filter on the conjunctive BM25F run). Surprisingly, it seems to be very difficult for users to actually identify useful phrases, there hardly is any improvement. In that sense, the findings from [MBSC97] seem to be still valid today.

In the light of these results, our second experiment aims at exploring if phrase queries have any potential at all for improving query effectiveness, i.e., how much can result quality be improved when the 'optimal' phrases are identified. Tables 5.8 and 5.9 show the precision at 10 results for our experiment with the different settings introduced in the previous section, separately for each TREC year.

topics	BM25F (conjunctive)	BM25F (disjunctive)	best user phrases	Wikipedia phrases
701-750 (TREC 2004)	0.536	0.548	0.546	0.566
751-800 (TREC 2005)	0.634	0.630	0.592	0.564
801-850 (TREC 2006)	0.528	0.538	0.546	0.526
average	0.566	0.572	0.561	0.552

Table 5.8: P@10 for different configurations and query loads, first part.

It is evident from the tables that an optimal choice of phrases can significantly improve over the result quality of the BM25F baseline, with peak improvements between 12% and 14% when term order remains unchanged, and even 17% to 21% when term permutations are considered[2]. Topics where phrases were most useful include "pol pot" (843), "pet therapy" (793) and "bagpipe band" (794) (which were usually identified by users as well). On the other hand, frequently annotated phrases such as "doomsday cults" (745) and "domestic adoption laws" (754) cause a drastic drop in performance. Interesting examples for improvements when permuting terms are "hybrid alternative

[2]both significant according to a paired t-test and Wilcoxon signed rank test, $p \leq 0.01$

topics	BM25F (conj.)	proximity score	best phrases	best phrases +permutations	best title-only TREC run
701-750 (TREC 2004)	0.536	0.574	0.616	0.668	0.588
751-800 (TREC 2005)	0.634	0.660	0.704	0.740	0.658
801-850 (TREC 2006)	0.528	0.578	0.606	0.654	0.654
average	0.566	0.604	0.642	0.687	0.633

Table 5.9: P@10 for different configurations and query loads, second part.

fuel cars“ (777) where the best phrase is actually “hybrid fuel” (with a P@10 value of 0.8, compared to 0.5 for the best in-order phrase and 0.2 for term-only evaluation in the form of BM25F (conjunctive)), and “reintroduction of gray wolves” (797) with P@10 of 1.0 with the phrase “wolves reintroduction”, compared to 0.6 otherwise (amongst others: reintroduction of “gray wolves”).

The best possible results are way above the best reported results for 2004 and 2005 and get close to the best result from 2006 (which was achieved, among other things, by the use of blind feedback)[3]. Wikipedia-based phrase recognition, a simple automated approach to phrase recognition, only leads to significant improvements for 2004 (paired t-test and Wilcoxon signed rank test, $p \leq 0.05$). For the remaining years we cannot observe significant improvements.

Interestingly, the proximity-aware score yields significant improvements over the baseline[4]; as it automatically considers “soft phrases”, there is no need to explicitly identify phrases here.

Discussion and Lessons Learned

The experimental analysis for phrase queries in this section yields the following results:

- We validated the common intuition that phrase queries can boost performance of existing retrieval models. However, choosing good phrases for this purpose is nontrivial and often too difficult for users, as the result of our user study shows.
- Existing methods for automatically identifying phrases can help to improve query performance, but they have their limits (like the methods based on Wikipedia titles evaluated here). While we expect that more complex methods (such as the advanced algorithm introduced in [Z+07]) will get close to the upper bound, they need to include term permutations to exploit the full potential of phrases. The common intuition that term order in queries bears semantics does not seem to match reality in all cases.
- Proximity-aware scoring models where the user does not have to explicitly identify phrases can significantly improve performance over a non-proximity-aware scoring model.

[3]no significance tests possible as we do not have per-topic results for these runs

[4]paired t-test and Wilcoxon signed rank test, $p \leq 0.1$ for TREC 2005 and $p \leq 0.01$ for the other two

Chapter 6

Top-k Vs. Non-Top-k Algorithms

This chapter starts with Section 6.1 that presents various top-k algorithms from the database systems community; they are classified according to the access methods to index lists required by the algorithms. Section 6.2 describes exact top-k algorithms (with and without term proximity component) and approximate top-k algorithms from the information retrieval community. The chapter concludes with Section 6.3 that explains some non-top-k algorithms.

6.1 Top-k Algorithms from DB

Top-k algorithms aim at efficiently assembling a ranked list of the k objects that match best the user need expressed by means of a top-k query. In the scenarios used throughout this thesis, objects may represent either elements of XML documents or full documents. To process top-k queries efficiently, a number of query processing techniques has been proposed over the last two decades (e.g., [Fag99, FLN03, CwH02, MBG04, GBK00, BGM02]).

To score an object, a *score aggregation function* F aggregates all known scores from different dimensions for this object. Assume that we want to aggregate scores for two objects o_i and $o_{i'}$. An aggregation function F is called monotone if $F(s_{i1}, \ldots, s_{im}) \leq F(s_{i'1}, \ldots, s_{i'm})$ when $s_{ij} \leq s_{i'j}$ for every dimension j, where s_{ij} and $s_{i'j}$ are the scores from dimension j for object o_i and $o_{i'}$, respectively.

The following descriptions assume that top-k algorithms have access to a set of m inverted lists $L = \{L_1, \ldots, L_m\}$ that represent one out of m score dimensions each. These algorithms assume that the scores for objects in each dimension j have been precomputed and stored in an inverted list L_j which is sorted by descending score, i.e., lists start with objects having high scores and end with objects having lower scores.

While a sorted access (also called sequential access) denotes an access to an object and its score during a sequential scan of a list, a random access denotes a direct access to an object and its score by the object identifier. Some algorithms use random lookups for promising candidates in dimensions where they have not yet been encountered; as such a *random access (RA)* is a lot more expensive than a *sorted access (SA)* (in the order of 50 to 50,000 according to [BMS+06]), an intelligent schedule for these RAs has a great impact on efficiency. The cost of one SA and one RA is denoted c_S and c_R, respectively.

In [Fag02] Fagin defines the middleware cost as

$$c_S \cdot \#SA + c_R \cdot \#RA$$

which corresponds to the query execution cost.

Algorithms from the family of *Threshold Algorithms* are similar to *dynamic pruning* approaches from the IR community. They start with a phase of sequential scans to each list involved in the query execution in an interleaved, round-robin manner. Processing lists in a *round-robin* manner characterizes the sequence in which lists are read: $(L_1, L_2, \ldots, L_m, L_1, \ldots)$, i.e., circular reads to each list, one after the other. As documents are discovered in this process, they are maintained as candidates in an in-memory pool, where each candidate has a current score also called *worstscore* (aggregated from the scores in dimensions where the document has been encountered so far). Additionally, each candidate object o_i has an upper score bound that is computed by setting all unknown scores to the highest possible score $high_j$ corresponding to the score at the current scan position (i.e., the last sequentially accessed tuple) of each list L_j:

$$bestscore(o_i) = F(p_{i1}, p_{i2}, \ldots, p_{im}),$$

where $p_{ij} = s_{ij}$ if o_i has been seen in L_j and $p_{ij} = high_j$ otherwise. p_{ij} is called predicate of object o_i in dimension j, $S(o_i)$ denotes those lists in L where o_i has been seen, $\bar{S}(o_i)$ those lists in L where o_i has not been encountered yet. A common choice for a monotonous aggregation function F is simple summation. Then, *bestscore* is defined as follows:

$$bestscore(o_i) = \sum_{j=1}^{m} \begin{pmatrix} s_{ij} & \text{if } L_j \in S(o_i) \\ high_j & \text{if } L_j \in \bar{S}(o_i) \end{pmatrix}. \tag{6.1}$$

To evaluate a top-k query, the algorithms typically maintain two priority queues: a priority queue (ordered by decreasing *worstscore*) that maintains a list of the k candidates with the highest *worstscore* values called the (intermediate) top-k results R, and another priority queue (ordered by increasing *bestscore*) that maintains the list of remaining candidates C that have the potential to qualify for the final R. The lowest *worstscore* of any object in R is named *min-k*. Candidates whose *bestscore* is not greater than *min-k* (i.e., the head of R), can be safely removed from C. The execution stops if all candidates in C have been eliminated and no unseen document can qualify for the final results; this is typically the case long before the lists have been completely read.

An excellent survey about top-k query processing techniques in the database systems area has been authored by Ilyas et al. [IBS08]. One way to classify these techniques is by the access methods to index lists required by the algorithms:

1. Sorted and random accesses to every list.

2. No random accesses, only sorted accesses.

3. Sorted accesses with carefully scheduled random accesses.

Marian et al. [MBG04] categorize sources (index lists) by their supported access methods: while *S-sources* provide sequential accesses only, *R-sources* provide random accesses only, and *SR-sources* provide both sequential and random accesses.

6.1.1 Sorted and Random Accesses

The algorithms described in this section use sorted as well as random accesses to lists. This means that all lists have to be SR-sources.

Fagin's algorithm (FA) [Fag99] proceeds in two rounds. In a first round, it performs sorted accesses to all lists in a round-robin manner until at least k objects have been fully evaluated. In a second round, for the remaining objects that have been encountered in at least one, but not all dimensions by sorted accesses, it performs random accesses to the lists representing the missing dimensions. Finally, the aggregation function F is applied to all seen objects and the objects are sorted such that the k objects with the highest scores can be returned.

The *Threshold algorithm (TA)* [FLN03] performs sorted accesses to all lists in a round-robin manner. When a new object o is seen by a sorted access to list L_x, TA performs random accesses to the remaining lists - therefore, it can compute the final score for o immediately. o is kept in the intermediate top-k results R iff it belongs to the k highest scores seen so far. C is always empty! After each round of sequential accesses, the $high_i$ values of the lists change, and the threshold value τ has to be updated: τ is calculated by combining the scores of the items read by the most recent sorted access to each list (i.e., $high_i$ for each list L_i) in a monotone aggregation function F. If τ underscores the *min-k* score (the lowest *worstscore* of the intermediate top-k results), the algorithm can safely terminate as no not yet seen object will be able to overscore the *min-k* score and thus make it to the top-k results R. The algorithm assumes that one sorted access has the same cost as one random access which is not valid in the cost model, but just influences TA's runtime behavior: this can lead to very expensive executions as the number of random accesses is not restricted and every sorted access can induce up to m-1 random accesses.

The *Quick-Combine algorithm* [GBK00] is a variant of TA. It uses an indicator $\Delta_i = \frac{\partial F}{\partial p_i} \cdot (S_i(d_i - c) - S_i(d_i))$ which estimates the utility to read from list L_i. The indicator considers 1) the influence of the predicate p_i used in list L_i on the overall score F and 2) the decay of the score in L_i over the last c steps which decreased the upper bound for not yet seen objects (i.e., $S_i(d_i - c) - S_i(d_i)$). The algorithm chooses the list with maximal Δ_i and works particularly well for skewed data.

Like FA and TA, the *Combined algorithm (CA)* [FLN03] performs sorted accesses to all lists in a round-robin manner. It makes use of the cost ratio between random and sorted accesses, $\gamma = \lfloor c_R/c_S \rfloor$: every time the depth of sorted accesses increases by γ, it picks the object o with missing information whose *bestscore* is largest and performs random accesses to all lists where o has not been encountered yet, short all lists in $\bar{S}(o)$. The algorithm can safely terminate if it has seen k objects and no object outside the top-k results R has a *bestscore* that overscores the *min-k* score and thus may make it to R. This includes the *bestscore* of the virtual document defined as $F(high_1, \ldots, high_m)$. The algorithm assumes that random accesses are more expensive than sorted accesses. Each of the picks for random accesses induces up to $m - 1$ random accesses as for TA; however random accesses are only triggered every γ sequential accesses. One can view CA as a merge between TA and NRA (cf. Section 6.1.2). If γ is very large (e.g., larger than the number of objects in all lists), CA corresponds to NRA. If $\gamma = 1$, CA is similar to TA: while CA performs RAs to all lists in $\bar{S}(o)$ for some object o, TA performs RAs to all lists in $\bar{S}(o)$ for every object o seen during round-robin sorted accesses.

6.1.2 No Random Accesses

Algorithms in this category only support sorted accesses and do not make use of random accesses. This means that all lists have to support sorted accesses (i.e, they are S-sources or SR-sources).

The *No Random Access algorithm (NRA)* [FLN03] performs sorted accesses to all lists in a round-robin manner. For each seen object, it keeps track of its *bestscore* and *worstscore* and the most recently seen scores $high_i$ per list L_i. The algorithm can safely stop when at least k objects have been seen and for all objects o that are not in the top-k results R (including the virtual document) holds $bestscore(o) \leq min\text{-}k$.

The *Stream-Combine algorithm* [GBK01] is similar to the NRA algorithm but favors sorted accesses to those lists that are more likely to lead to early termination than others. To estimate the utility of reading next from list L_i, it uses an indicator similar to the one used in the Quick-Combine algorithm $\Delta_i = \#M_i \cdot \frac{\partial F}{\partial p_i} \cdot (S_i(d_i - c) - S_i(d_i))$. This indicator also considers the cardinality of M_i, the subset of the intermediate top-k results R whose *bestscore* would be decreased or whose precise score would be known after reading from list L_i.

6.1.3 Carefully Scheduled Random Accesses

Algorithms in this category require that at least one list is sequentially accessible (i.e., the list is an S-source or an SR-source) to get an initial set of candidate objects that may make it into the final top-k results. The Upper and Pick algorithms [BGM02, MBG04] have been proposed in the context of Web-accessible sources categorized by their supported access methods.

The *Upper algorithm* [BGM02] fills the *bestscore*-ordered candidate queue C using round-robin sorted accesses to sorted sources (S-sources and SR-sources). In each round, it checks whether C has run empty or the object o_{top} with the highest *bestscore* in C underscores the threshold τ defined as $F(high_1, \ldots, high_m)$ (i.e., the *bestscore* of an unseen document). If one of these two conditions holds, the Upper algorithm performs a sorted access and inserts the read object o into the candidate queue C or updates o's *bestscore* (if it has already been in the queue). Given the new $high_i$ value,

τ can be updated. If the score of o is final (i.e., $bestscore(o) = worstscore(o)$), o is returned as a member of the top-k results. If none of these two conditions holds, the algorithm selects the best source for o_{top} to perform a random access which can come in different implementations. The algorithm stops iff k objects have been returned.

The *Pick algorithm* [MBG04] chooses the object o with the largest difference between $worstscore(o)$ and $bestscore(o)$ to perform a random access. The source to be probed for o is randomly chosen from the set of sources that represent a score dimension not yet known during the evaluation of o.

The *Minimal Probing (MPro) algorithm* [CwH02] works in two phases, 1) the initialization phase which performs only sorted accesses and 2) the probing phase which performs random accesses to complete scores. The initialization phase uses sources that provide sorted access to fill the candidate queue C with objects. Before inserting each object, the MPro algorithm assigns a $bestscore$ value to the object that considers the maximum scores of the remaining, expensive sources representing unknown score dimensions. In each iteration, the probing phase removes the object o with the highest $bestscore$ from the candidate queue and probes its next unevaluated source. If the evaluation for o is complete, o is returned as part of the top-k results, otherwise it is reinserted into the candidate queue. The algorithm stops as soon as k objects have qualified for the top-k results. Finding the optimal probing schedule for each object is an NP-hard problem: thus, optimal probing schedules are approximated using a greedy approach that relies on benefit and cost of each predicate obtained by sampling of ranked lists at query startup time. The authors prefer global scheduling (i.e., the probing sequence is the same for every object) since per-object scheduling would generate the N-fold cost, given N objects in the database. In contrast to the Upper algorithm, MPro expects as input a fixed schedule of accesses to R-sources fixed during the initial sampling phase [MBG04]. Thus, during query processing, it selects only the object to probe next, but avoids source selection at run time that is necessary for Upper. Assuming that a sorted access is cheap, this algorithm aims at minimizing the cost of random accesses.

IO-Top-k [BMS$^+$06] processes index lists in batches of b sorted accesses which are distributed across the index lists. The authors reuse the cost model (middleware cost) introduced by Fagin in [Fag02] which makes $c_S \cdot \#SA + c_R \cdot \#RA$ the overall objective function.

The goal of *SA scheduling* is to optimize in each batch the individual batch sizes b_i across all lists, such that some benefit function is maximized and $\sum_{i \in \{1,\ldots,m\}} b_i = b$ which is equivalent to solving the NP-hard knapsack problem. The authors propose two strategies to handle the problem. The *Knapsack for Score Reduction (KSR) method* aims at reducing $high_i$ values as quickly as possible as low $high_i$ values allow earlier candidate pruning. Given the current scan positions in the index lists and a budget b, the goal is to find a schedule of individual batch sizes per list such that the total expected reduction of $bestscore$ values for all candidates is maximized. $bestscore$ values are estimated using histograms, assuming uniformly distributed scores. Besides, the optimization expects that the probability of seeing a particular document in list L_i

where it has not been encountered yet is (close to) zero as only a small part of a list (i.e., b_i entries) is scanned in the next batch. Thus, the expected reduction of that document's *bestscore* corresponds to the estimated delta (with the help of histograms) in $high_i$ for b_i read entries in L_i. The *Knapsack for Benefit Aggregation (KBA) method* performs typically better than KSR, uses the notion of benefit per candidate, and aggregates all candidates' benefits to decide about b_i choices. The goal is to achieve low SA costs in the overall objective function. Like KSR, it uses histograms and current scan positions but also uses knowledge (obtained by the scans so far) about the candidate under view.

The goal of *RA scheduling* is 1) to increase *min-k* in the beginning and 2) clarify scores for candidates to allow early termination in later stages. The authors propose two strategies coined Last-Probing and Ben-Probing. *Last-Probing* proceeds in two phases: the first phase consists of several rounds of SAs, the second phase performs only RAs. The second phase is started iff 1) the expected cost for RAs is less than the number of all SAs done up to that point and 2) $\sum_{i=1}^{m} high_i \leq min\text{-}k$. The first condition aims at balancing the costs of SAs and RAs, the second condition ensures that all top-k items have been seen at this point. *Ben-Probing* uses a probabilistic cost model to compare the benefit of performing RAs against performing SAs. Costs are compared every b steps when SA scheduling has to be done anyway.

The most efficient variant, coined `RR-LAST` mode, does round-robin sequential accesses and schedules all RAs only at the end of the partial scans of inverted lists, namely, when the expected cost for RAs is below the cost of all sequential accesses so far.

6.2 Top-k Algorithms from IR

Besides top-k algorithms from the database systems area (cf. Section 6.1), there are also approaches for top-k query processing in the information retrieval domain (e.g., [AM06, BCH$^+$03b, SC07, TMO10, DS11]).

Following the classification by Ding and Suel [DS11], indexes can be organized in different ways:

- document-sorted: the postings in each inverted list are sorted by docid.
- impact-sorted: the postings in each inverted list are sorted by their impact, i.e., their influence on the score of a document which assumes that the scoring function is decomposable (i.e., one can sum up contributions of single term entries).
- impact-layered: the postings are organized in layers, with postings in layer i having a higher impact than the ones in layer $i+1$. Each layer's entries are sorted by docid.

In impact-sorted and impact-layered indexes the postings with the highest impact can be found at the start of the inverted lists such that they are read first during query processing. This property makes impact-sorted and impact-layered indexes popular for early termination algorithms. Impact-sorted indexes cannot use docids for compression since docids increase and decrease with decreasing impact. They can be compressed if the number of distinct impacts is small or if small integer numbers are used as impacts. Impact-layered indexes which employ a small number of layers may be better to compress, but do not reach the same compression level as document-sorted indexes whose docid gaps are smaller. Only very few early termination techniques use document-sorted index structures.

The IR community usually categorizes index traversal approaches as follows:

- DAAT (document-at-a-time): the postings of document d_j are processed before the postings of document d_{j+1}. Each document is assigned a final score before the next document is scored and a set of the k documents with the currently highest scores is maintained.
- TAAT (term-at-a-time): the inverted list of query term t_i is processed before the inverted list of query term t_{i+1}. Documents' partial scores are maintained in so-called accumulators which keep both candidates C and (intermediate) top-k results R. As TAAT approaches do not know the final scores of objects immediately, but maintain partial scores instead, the memory footprint is larger for TAAT than for DAAT approaches.
- SAAT (score-at-a-time): this approach is neither strictly TAAT nor DAAT. All inverted lists are open at the same time and pointers to list entries that make larger contributions to document scores are processed first. It requires the index structures to be organized in an impact-sorted or impact-layered form.

6.2.1 Exact Top-k Algorithms from IR

This subsection elaborates on top-k algorithms from the IR community that deliver exact top-k results, but do not involve proximity scores.

Anh and Moffat [AM06] propose dynamic pruning methods that use impact-layered indexes. For each document, all queryable terms are sorted by decreasing term frequency value within that document. Given the ordering, each term in the document is assigned an impact between an upper limit u (decided at indexing time) and 1. The number of terms assigned to lower valued layers is exponentially growing. Stop words are always assigned the lowest impact 1. Considering a document with n_d distinct terms, of which n_s stop words, the base of the layer is defined as $B = (n_d - n_s + 1)^{1/u}$. The layers contain $(B-1)B^i$ items, where $i \in \{0, \ldots, u-1\}$; in [AM06] Anh and Moffat choose $u = 8$. This approach allows a high compression and storage of documents in impact order. The index structure resembles the inverted block-index structure used by Bast et al. for *IO-Top-k* [BMS+06] (Section 6.1). Bast et al. partition each index list into blocks, which are ordered by descending score. Within each block, index entries are stored in item ID order which is comparable to document-sorted partial indexes.

The authors describe a pruning method that proceeds in four stages and relies on an SAAT approach with an impact-layered index. The algorithm maintains accumulators that keep track of candidate documents C (and their worstscores) which may qualify for the top-k results R. Furthermore, it keeps track of the current top-k results R, the $min\text{-}k$ score (i.e., the lowest worstscore among the items in R, $worstscore(R_k)$). R_i denotes the result at rank i, i.e., the result with the i^{th} largest score.

For each list L_i, the impact of the next not yet processed document in the inverted list is stored as $next_i$. As aggregation function Anh and Moffat use the commonly used simple summation.

The algorithm runs in four phases: the initial *OR phase* accepts new candidates to be added. The subsequent *AND phase* only updates already existing candidates and top-k results but does not add new ones. The *REFINE phase* only considers documents that are in the top-k results and reorders them. The final *IGNORE phase* ignores the remainder of all inverted lists. The initial *OR phase* can be quit if no document that is not yet in C or R (i.e., that has not yet been read in an inverted list), can make it to the final top-k results R which holds if

$$min\text{-}k \geq \sum_{L_i \in L} next_i.$$

This criterion corresponds to the stopping criterion for the virtual document o applied in the NRA (Section 6.1.2): $bestscore(o) \leq min\text{-}k$ with $bestscore(o) = F(high_1, \ldots, high_m)$ and $high_j = next_j$. The subsequent *AND phase* can be left when the set of top-k results will not change any more:

$$min\text{-}k \geq max\{bestscore(d) : d \in C, d \notin R\},$$

where

$$bestscore(d) = worstscore(d) + \sum_{L_j \in \bar{S}(d)} next_j.$$

This criterion corresponds to the stopping criterion for all candidate objects applied in the NRA. The *REFINE phase* can be stopped when the sequence of the top-k results will not change any more. This holds if for all top-k documents the bestscore of the document at rank i is not larger than the worstscore of the document at rank $i-1$:

$$\forall R_i, R_{i-1} \in R, i \leq k : bestscore(R_i) \leq worstscore(R_{i-1}).$$

The final *IGNORE phase* can ignore all remaining postings then. In addition, the authors propose a method which limits the number of entries read after the *OR phase* and reaches precision@20 values comparable to the algorithm just described, already when stopping reading inverted lists after 30% of the entries that have not been read during the *OR phase.* The method features low memory requirements and is way faster than exhaustive evaluation since only needed fragments of inverted lists are transferred from disk.

Strohman and Croft [SC07] keep the entire index in main memory to avoid expensive random accesses to disk such that their query processing cost is determined by the number of read bytes. The impact-layered index uses the same impact model as Anh and Moffat, with only eight different integer valued term weights [AM06]. The index is organized in segments where every segment contains a set of documents sharing the same impact value. Each segment is document-sorted.

The algorithm is based on Anh and Moffat's approach adding some optimizations: while Anh and Moffat's algorithm prunes candidates only once, once the top-k results are known (just before the *REFINE phase* starts), Strohman and Croft eliminate candidates after each inverted list segment has been processed: document d can be removed from the candidates if $min\text{-}k \geq bestscore(d)$; doing so, the number of candidates to be updated can be reduced earlier.

In addition, the authors propose a technique to optimize the list-length dependent skipping distance such that inverted list skipping can be applied during both the *AND* and the *REFINE phase.* As both accumulators and inverted list segments are document-sorted, large sections in the inverted lists not worth decoding can be identified and skipped.

The *WAND approach* devised by Broder et al. [BCH+03b] uses a document-sorted index with DAAT-based query processing. It maintains a list of top-k items R scored so far, sorted by decreasing score, and sets the threshold τ to the score of the k^{th} item, R_k. (Scores of items are always complete scores, i.e., $bestscore{=}worstscore$.) Furthermore, for each inverted list, the algorithm keeps track of the current scan position as well as the maximum score in the respective list.

The algorithm uses pivoting in order to skip postings and proceeds in multiple iterations: at the beginning of each iteration, the pointers to the inverted lists are ordered by ascending current docid. Then the inverted lists' maximum scores are aggregated (in the sequence of the ordered pointers), one after the other, until τ is exceeded. The term corresponding to the inverted list where τ has been exceeded is called *pivot term*, and the current document in the respective inverted list is called pivot document; the pivot document has the smallest docid with the chance to exceed τ. However, the pivot document is only valid, if the current docids of all preceding inverted lists are equal to the pivot docid. Then the corresponding document can be scored. Otherwise, the cursor of one of the preceding term lists is moved to the pivot docid and the next iteration can start.

Ding and Suel's *Block-Max WAND (BMW) algorithm* [DS11] is a state-of-the-art DAAT algorithm that uses dynamic pruning and which is based on the WAND algorithm. The focus of [DS11] is on top-k early termination query processing (in the sense of non-exhaustive evaluation) using main-memory based index structures.

The authors devise an inverted index structure called *block-max index* which sorts the inverted list in docid order like the input for the WAND algorithm, but organizes the compressed inverted list in blocks. For each block, it stores the maximum impact score of a posting in the block in an uncompressed form to allow skipping long list parts. The inverted lists' block size is 64 or 128 documents (postings) and supports decompressing individual blocks. There is an additional table outside the inverted list blocks (to avoid cache line effects) that stores the maximum (or minimum) docid and the block size. Storing this extra information only slightly increases the index size.

For the WAND algorithm, skipping is limited because the inverted indexes only store the maximum impact score of the entire list. Storing the maximum impact score per block for block-max indexes enhances the skipping known from the WAND algorithm. This way, the upper bound approximation of document impact scores can be lowered and large performance improvements achieved.

The authors distinguish between *deep pointer movement* in an inverted list which usually involves block decompression and *shallow pointer movement* which moves the current pointer to the corresponding block without decompressing the block. To this end, the shallow pointer movement relies on the block boundary information stored in the additional table.

As stated before, the BMW algorithm is based on WAND and thus uses pivoting in order to skip postings and proceeds in multiple iterations: before evaluating a pivoting docid, first the shallow pointers are moved to check whether the document can make it to the top-k based on the maximum score per block. If not, another candidate is chosen. Instead of moving the cursor in one list to pivot docid+1, Ding and Suel choose $d' = min\{c_1, \ldots, c_{p-1}, c_p\}$, where c_1 to c_{p-1} are the block boundaries plus one of the first $p-1$ lists and c_p is the current docid in the pivot term list. This approach greatly improves skipping compared to moving the cursor in one list to pivot docid+1.

Ding and Suel experimentally validate that the basic BMW algorithm outperforms their implementation of Strohman and Croft's approach which is again faster than the WAND approach.

Extensions include an impact-layered index organization and docid reassignments. The idea of docid reassignments is to give similar web-pages close docids to improve index compression. Ding and Suel attribute docids by alphabetical ordering of the URLs as in their earlier work with Yan [YDS09b]; it seems that after reassignment, documents in the same block tend to have more similar scores which in addition helps to speed up query processing. The average speed is still slightly slower than for exhaustive conjunctive evaluation, but the difference is greatly narrowed.

The goal of impact-layered index organization here is to put high-scoring documents in the same layer and thus avoid spiky scores in the remaining layers. This approach is supposed to avoid reading as many blocks in the remaining layers as possible during the execution of the BMW algorithm. The impact-layered index is split into N layers and each layer is treated like a separate term with the disadvantage of the larger number of terms per query. Ding and Suel choose N=2 to avoid decreasing performance, only lists with at least 50,000 postings are split with 2% of the postings added to the first layer. That way runtime can be decreased further, almost meeting the average speed of exhaustive conjunctive evaluation.

6.2.2 Exact Top-k Algorithms from IR with a Term Proximity Component

This subsection presents approaches that deliver exact top-k results and incorporate some term proximity scoring into dynamic pruning. There has been little work published that incorporates proximity scores to accelerate top-k text retrieval. [ZSLW07, ZSYW08, YSZ+10] use a combination of term as well as term proximity scores similar to the solution presented later in Chapter 7 and make additional use of PageRank scores. [ZSLW07] propose a PageRank ordered index structure that segments index entries based on the tags that surround the text the index entry has been generated from. This results in a long body tag segment and a short segment for all remaining tags. An extension of this index structure splits the body tag segment in two segments, based on whether a document's term weighting score exceeds or underscores a threshold score. A very recent approach to use term pair indexes to improve bounds in top-k text retrieval was presented in [YSZ+10] which focuses on the index building and query processing with term-pair indices on every local machine of a cluster. They use an order-aware proximity score resulting in two term pair lists per term pair. Like [ZSLW07, ZSYW08], the approach is only applicable for two-term-queries. To keep the ranking flexible [YSZ+10] store position information while we make use of an integrated proximity score (cf. Section 7.2.2). Our approximate approach that we will present in Chapter 8 elaborates on the trade-off between index size and result quality which was not mentioned in [YSZ+10]. In contrast to [ZSLW07, ZSYW08, YSZ+10, SBH+07, BS12], [TMO10] do not create term pair lists statically but dynamically during query processing to save on disk space. Document-sorted term pair posting lists are generated from two document-sorted single term lists by a merge join-based operation. To save on I/O operations, only single term lists are read from disk and decompressed: i.e., the pointer of the list with the minimum current docid in two document-sorted single term lists is moved. If the two pointers point to the same document, this document qualifies for the term pair list. The authors analyze two existing DAAT dynamic pruning strategies, MaxScore (terminates one item's scoring if its score cannot exceed *min-k*) and WAND (cf. Section 6.2), and modify them to support proximity scores. They accelerate MaxScore and WAND in a two-stage approach: in a first stage, only single term posting lists are processed like in WAND or MaxScore. In a second stage, term pairs are subsequently processed using early termination (with the MaxScore strategy).

The order-aware proximity score uses the sequential term dependence (SD) model for Markov Random Fields [MC05] (cf. Section 2.7.3). For all pairs of adjacent query terms it captures the number of exact phrase occurrences and term pair occurrences in a text window of size 8 for a document d. In [MOT11], the upper bound of a term pair (t_i, t_{i+1})'s frequency is approximated by the maximum term frequency in the term posting lists $L(t_i)$ and $L(t_{i+1})$: $min(max_{d \in L(t_i)}(tf(t_i, d)), max_{d \in L(t_{i+1})}(tf(t_{i+1}, d)))$ as no term pair can occur more often in a document than the least frequent of the constituent terms. If a term pair posting is selected for scoring, the exact term pair frequency for a text window size is computed using position lists in both single term posting lists. Otherwise, this computation can be avoided.

6.2.3 Approximate Top-k Algorithms from IR

While the approaches described above compute the exact top-k results for a scoring model with queries on an indexed collection, the approaches described in this subsection just approximate the top-k results instead which is often good enough in terms of result quality.

Approximate top-k algorithms include probabilistic result pruning [TWS04], execution with limited budget [SSLM$^+$09], and improving score bounds for proximity scores by means of pruned bigram indexes [ZSYW08].

In contrast to dynamic pruning approaches which maintain full index lists and evaluate only a fragment of the indexed documents at query processing time, static pruning approaches such as [SCC$^+$01, BC06] discard postings considered not important already at indexing time. This incurs less stored information on hard disk and often opens the opportunity to keep the indexes in memory of a single machine which saves on I/O time during processing. If indexes are wisely pruned, the retrieval quality of the top-k results is comparable to dynamic pruning approaches, usually at the expense of lower recall values. [SCC$^+$01] introduced list pruning with quality guarantees for the scores of query results, assuming top-k style queries with a fixed (or at least bounded) k. For each list L, they consider the score $s_k(L)$ at position k (the k^{th} highest score) in L, and drop each entry from that list whose score is below $\epsilon \cdot s_k(L)$, where $0<\epsilon<1$ is a tuning parameter. They assume a given k, ϵ, and the original score S that uses unpruned lists as input. They prove that for each query q with $r < \frac{1}{\epsilon}$ terms there is a scoring function S' such that for every document

$$(1 - \epsilon r)S(q, d) \leq S'(q, d) \leq S(q, d).$$

S' is similar to a scoring function on pruned lists except for the case that a document's entries have been pruned away in too many dimensions such that its score becomes zero. Experiments are carried out with the topics 401-450 from the Ad Hoc Task of TREC-7 in short and long variants: short queries use the titles only, whereas long queries use titles and descriptions. Choosing $\epsilon = 50\%$ provides similar P@10 values as on the unpruned index.

[BC06] prune lists using a document-centric approach. The approach decides, based on a term's contribution to a document's Kullback-Leibler divergence from the text collection's global language model, whether the corresponding posting should remain in the index. For each document d in the text collection their best-performing approach ($DCP^{(\lambda)}_{Rel}$) keeps only the postings for the top-k_d terms in d, where $k_d = \lceil dt(d) \cdot \lambda \rceil$ and λ is a user-defined pruning parameter. Using a pruned index with $\lambda = 0.1$ (i.e., 10% of each document's terms are kept) generates a result quality slightly worse than using an unpruned BM25 index evaluated with the 50 Ad Hoc topics from the TREC 2005 Terabyte Track. Its size of 1,570MB corresponds to 12% of the size of an unpruned index. Given a fixed response time, $DCP^{(\lambda)}_{Rel}$ can provide a better result quality than two other strategies at most recall levels on the TREC 2004 and 2005 Terabyte Track test beds. The first strategy indexes a constant number of terms per document and the second strategy performs term-centric pruning which keeps the k best postings for the n most frequent terms.

[ZSYW08] combine static index pruning with dynamic pruning techniques. They use pruned uncompressed bigram indexes derived by static index pruning as an additional input to dynamic pruning top-k processing. The pruned bigram indexes lower the upper bound for term proximity scores during query processing of two-term-queries. The pruning technique discards 1) bigrams when they are rare in the collection (as that two-term query is unlikely to be issued) and 2) bigrams when both terms are rare in the collection (inducing short term indexes which can be processed quickly). A combination of the pruned bigram index and a two-segment index (i.e., high and low score segment), with each segment ordered by PageRank score, processes retrieved results most efficiently.

[WLM11] propose a cascade ranking model, a sequence of increasingly complex ranking models. The first stage returns the highest scoring documents according to the first applied scoring model. Each subsequent stage first prunes candidates and then refines the scoring for the remaining candidates used as input to the next stage. The authors propose rank-based, score-based, and score distribution-based prunings. Unigram features and bigram proximity features (both ordered and unordered term occurrences) as proposed in [MC05] are integrated into a Dirichlet and BM25 score, respectively. A boosting algorithm (based on AdaRank [XL07]) learns the cascade sequence and feature weights of the individual scoring functions. To this end it uses a tradeoff metric that weights effectiveness and efficiency (costs). The cost of a scoring model depends on the normalized average run time over a set of training queries and the input size of this stage.

6.3 Non-Top-k Algorithms

The highly efficient top-k or dynamic pruning algorithms (cf. Sections 6.1 and 6.2) that are frequently applied for efficient query processing incur a non-negligible processing overhead for maintaining candidates and candidate score bounds, for mapping newly read index entries to a possibly existing partially read document using hash joins, and for regularly checking if the algorithm can stop. In scenarios with short index lists, this processing overhead is not necessary. Instead, it is sufficient to exhaustively evaluate queries in a DAAT fashion. If the lists are long, one should prefer a top-k algorithm instead.

The *n-way merge join* is a DAAT algorithm which receives n docid-ordered lists as input and in each join step calculates the (full) score for the document $d_{current}$ having the next smallest not yet evaluated docid. If the algorithm is executed in the *exhaustive OR* mode (disjunctive query evaluation), $d_{current}$ does not have to be seen in every list. If the algorithm is executed in the *exhaustive AND* mode (conjunctive query evaluation), the current docid must be seen in every list and the computation of $d_{current}$'s score can be skipped if one list pointer points to a different document. If the score is higher than the *min-k* value, the document is kept in a heap of candidate results, otherwise it is dropped as it cannot make it into the top-k results R any more. For every list it keeps track of the position up to which the list has been read so far and iterates to the next item if the document in this list has just been evaluated. If the items of all lists have been read completely, the algorithm terminates. Once all index entries have been read, the content of the heap is returned.

One commonly used approach to accelerate query processing is to perform ranking in two phases. The first phase that uses a simple and easily-to-compute ranking model (e.g., BM25) pre-selects the documents to be re-ranked in the second phase with a usually more complex, not that easily-to-compute scoring model. Furthermore, using phrases is a common means in term queries to restrict the results to those that exactly contain the phrase and is often useful for effective query evaluation [CCB95]. A simple way to efficiently evaluate phrases are *word-level indexes*, inverted files that maintain positional information [WMB99]. There have been some proposals for specialized index structures for efficient phrase evaluation that utilize term pair indexes and/or phrase caching, but only in the context of boolean retrieval and hence not optimized for top-k style retrieval with ranked results [CP06, W^{+}99, WZB04]. There are proposals to extend phrases to window queries, where users can specify the size of a window that must include the query terms to favor documents containing all terms within such a window [MSTC04, PA97, BAYS06]. However, this line of works has treated term proximity only as an afterthought after ranking, i.e., proximity conditions are formulated as a simplistic Boolean condition (e.g., requiring query terms to appear within the user-specified window size) and optimized as separate post-pruning step after ranked evaluation.

Chapter 7

Casting Proximity Scoring Models into Top-k Query Processing

7.1 Introduction

The first part of this chapter describes how we can modify Büttcher et al.'s scoring model to make it fit into top-k algorithm-based query processing. It is based on our work published in [SBH$^+$07].

There has been a number of proposals in the literature for proximity-aware scoring schemes summarized in Chapter 2; however, there are only a few proposals that efficiently find the best results to queries in a top-k style with dynamic pruning techniques (cf. Section 6.2.2). We show that integrating proximity in the scoring model can not only improve retrieval effectiveness, but also improve retrieval efficiency; using pruned index lists, we gain up to two orders of magnitude compared to standard top-k processing algorithms for purely occurrence-based scoring models on unpruned lists.

This insight opens the door for using a light-weight n-ary merge join in combination with pruned document-sorted index lists published in [BS08a] which realizes a similar speed up by one or two orders of magnitude compared to an evaluation with a top-k system such as TopX [TSW05] using unpruned lists. Hence, we can avoid top-k dynamic pruning techniques that maintain a candidate pool and compute best-/worstscores for result candidates to finally come up with the top-k results. Besides saving the overhead costs, this simple approach keeps up the excellent precision values and saves much disk space.

The second part of this chapter aims at evaluating the feasibility of the proximity-enhanced scoring models surveyed in Chapter 2 for top-k algorithm-based query processing. Thereby, we try to figure out how to apply the techniques presented in the first part of this chapter to other scores.

7.2 Proximity Scoring

7.2.1 Proximity Scoring Models

We focus on proximity scoring models that use a linear combination of a content-based score with a proximity score as they are usually more easily decomposable into their features and thus more straight forward to index than integrated scoring models. We have described a selection of such linear combination and integrated scoring models in Section 2.4 and Section 2.5, respectively.

The particular scoring model we use is a scoring model proposed by Büttcher et al. [BC05, BCL06] (labelled *Büttcher's scoring model* from now on) which has been described in detail in Section 2.4.2. We have experimentally validated in Section 4.2 that for the Web Track and Robust Track test beds, Büttcher's scoring model is among the scoring models that provide the highest precision, MAP, and NDCG values. For the Terabyte Track test beds, it yields the highest retrieval quality for all test beds and retrieval metrics (except for topics 751-800 with the MAP metrics where it performs slightly weaker than Song et al.'s scoring model) and for all INEX test beds the highest retrieval quality with all metrics. According to Metzler [Met06b], an ideal model that generalizes perfectly achieves an effectiveness ratio of 1. While effectiveness ratios below 90% indicate a scoring model's missing ability to generalize, the most reasonable retrieval models have an effectiveness ratio above 95%. In Section 4.2.3 we have demonstrated for the MAP and NDCG@10 metrics that Büttcher's scoring model has an effectiveness ratio that overscores 95% which holds only for two scores in our evaluation. All scoring models exhibit high intracollection generalization values between 98% and 100% on all test beds with both the MAP and NDCG@10 metrics. In Section 4.2.4, we have shown for both the MAP and NDCG@10 metrics that Büttcher's scoring model exhibits a relatively low spread, but a relatively high entropy. In our setting, we think that the spread value is more meaningful than the entropy value as it measures how much retrieval quality can decrease if we choose the wrong parameter combination.

An initial set of experiments aimed at validating that Büttcher's score outperformed the BM25 score for various parameter settings and thus shows improvements independent of the parameter choice. In particular, we wanted to find out whether the original parameter setting from [BC05, BCL06] is appropriate and can be used for our experiments. To this end, with the 100 topics from the TREC Terabyte Track, Ad Hoc Tasks 2004 and 2005 on the GOV2 collection, we evaluated the effect of Büttcher's score over the BM25 score alone for 60 combinations of values for k_1 and b, for precision at different cutoffs and MAP. For all experiments, the results with Büttcher's score were always at least as good as the results with BM25, significantly better (with $p \leq 0.05$ for a signed t-test) for 42 configurations in precision at 10 results, for 59 configurations in precision at 100 results, and always for MAP. We use the parameter setting from [BC05, BCL06] ($k_1 = k = 1.2, b = 0.5$), which was among the best configurations in our experiments as well.

7.2.2 Modification of Büttcher's Scoring Model

To include Büttcher's proximity score into query processing, it would be intriguing to use a standard word-level inverted list, i.e., an inverted list that stores with each document also the positions of the term occurrences in the document, and compute proximity scores on the fly as a document is encountered. We could use the $tf(t,d)$ values for each query term t in d to compute a bestscore for a document: to this end we would have to 'construct' a document that maximizes the $pscore(d,q)$ value by putting $tf(t,d)$ times query term t into the conceived document (we do not know the real document since we have not read the word-level inverted lists yet). This boils down to a combinatorial problem. For two-term queries $\{t_i, t_j\}$ it is already hard to solve; if t_i and t_j share the same tf value in d, one has to place them alternately in the conceived document to maximize the *pscore* value as only non-equal adjacent query terms generate a proximity contribution. If t_i and t_j have different tf values in d, we first have to place the term with the lower tf value (w.l.o.g. t_i) and then try to group the term occurrences of t_j around the occurrences of t_i. The longer the query, the more complex the combinatorial problem gets. However, this approach is not feasible in a top-k style processing as it is not possible to compute *tight* score bounds for candidates which in turn disables dynamic pruning and in addition the combinatorial problem does not seem to be trivial, especially for long queries.

For an efficient computation of the top-k results, we need to precompute and store proximity score information in index lists that can be sequentially scanned and compute tight score bounds for early termination. The main problem with Büttcher's scoring function in this respect is that the accumulator value $acc_d(t)$ is computed as a sum over *adjacent* query term occurrences, which is inherently query dependent, and we cannot precompute query-independent information. An additional, minor issue is that the scoring function includes the document length which cannot be easily factorized into a precomputed score contribution.

To solve this, we slightly modify Büttcher's original scoring function; this does not have much influence on result quality, but allows precomputation. In addition to dropping the document length, by setting $b = 0$ in the formula, we consider *every* query term occurrence, not only adjacent occurrences. The modified accumulation function acc' is defined as

$$acc'_d(t_k) \quad = \sum_{\substack{(i,j) \in Q_{all,d}(q): \\ p_i = t_k, p_i \neq p_j}} \frac{idf(p_j)}{(i-j)^2} + \sum_{\substack{(i,j) \in Q_{all,d}(q): \\ p_j = t_k, p_i \neq p_j}} \frac{idf(p_i)}{(i-j)^2}. \tag{7.1}$$

As the value of $acc'_d(t_k)$ does not only depend on d and t_k, but also on the other query terms, we still cannot precompute this value independently of the query. However, we can reformulate the definition of $acc'_d(t_k)$ as follows:

$$acc'_d(t_k) = \sum_{t \in q} idf(t) \underbrace{\left(\sum_{\substack{(i,j) \in Q_{all,d}(q): \\ p_i = t_k, p_j = t, \\ p_i \neq p_j}} \frac{1}{(i-j)^2} + \sum_{\substack{(i,j) \in Q_{all,d}(q): \\ p_i = t, p_j = t_k, \\ p_i \neq p_j}} \frac{1}{(i-j)^2} \right)}_{:= acc_d(t_k,t)} \tag{7.2}$$

$$= \sum_{t \in q} idf(t) \cdot acc_d(t_k, t). \tag{7.3}$$

We have now represented $acc'_d(t_k)$ as a monotonous combination of *query term pair scores* $acc_d(t_k, t)$. We can precompute these pair scores for all term pairs occurring in documents and arrange them in index lists that are sorted by descending $acc_d(t_k, t)$ scores. Note that term order does not play a role, i.e., $acc_d(t_k, t) = acc_d(t, t_k)$. Including these lists in the sequential accesses of our processing algorithm, we can easily compute upper bounds for $acc'_d(t_k)$ analogously to query term dimensions by plugging in the score at the current scan position in the lists where d has not yet been encountered. The current score of a document is then computed by evaluating our modified Büttcher score with the current value of acc'_d, and the upper bound is computed using the upper bound for acc'_d; this is correct as the modified Büttcher score is monotonous in acc'_d.

7.3 Indexing and Evaluation Framework

7.3.1 Precomputed Index Lists and Evaluation Strategies

Our indexing framework consists of the following precomputed and materialized index structures, each primarily used for sequential access, but with an additional option for random access:

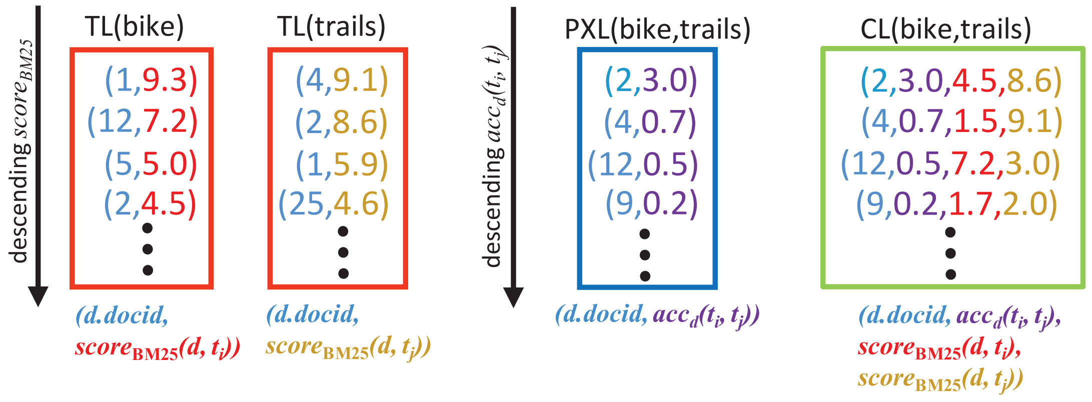
- `Term index list` (short: term list): for each single term t a list that contains an entry for each document d where this term occurs (i.e., $tf(t, d) > 0$). This entry has the form $(d.docid, score_{\text{BM25}}(d, t))$ where $d.docid$ is a unique numerical id for document d. TL(t) denotes the term list of term t. The chosen parameters have been disclosed in Section 7.2.1.

- `Proximity index list` (short: proximity list): for each single term pair (t_1, t_2) a list that contains an entry for each document d where this term pair occurs within any text window of size W (we will discuss the window size in Section 7.3.4). This entry has the form $(d.docid, acc_d(t_1, t_2))$ where the proximity contribution of (t_1, t_2) for d is stored in $acc_d(t_1, t_2)$. t_1 and t_2 are lexicographically ordered (i.e., $t_1 < t_2$) such that for any single term pair combination we keep the corresponding proximity list just once. PXL(t_1, t_2) denotes the proximity list for the term pair (t_1, t_2).

- `Combined index list` (short: combined list): for each single term pair (t_1, t_2) a list that contains an entry for each document d where this term pair occurs within any text window of size W (we will discuss the window size in Section 7.3.4). This entry has the form $(d.docid, acc_d(t_1, t_2), score_{\mathrm{BM25}}(d, t_1), score_{\mathrm{BM25}}(d, t_2))$ where the proximity contribution of (t_1, t_2) for d is stored in $acc_d(t_1, t_2)$. t_1 and t_2 are lexicographically ordered (i.e., $t_1 < t_2$) such that for any single term pair combination we keep the corresponding combined list just once. CL(t_1, t_2) denotes the combined list for the term pair (t_1, t_2).

Both PXLs and CLs are term pair lists (short: pair lists). The order of entries in the index lists depends on the algorithm used for query processing. Entries can be ordered either by docid or by descending scores ($score_{\mathrm{BM25}}$ for the term lists, acc_d values for the term pair lists).

We illustrate the layout of our index lists with score-based ordering in Figure 7.1. It depicts the term, proximity, and combined index lists which can be used to process the query $\{bike, trails\}$.



descending $score_{BM25}$
TL(bike)
(1,9.3)
(12,7.2)
(5,5.0)
(2,4.5)
(d.docid, $score_{BM25}(d, t_i)$)
TL(trails)
(4,9.1)
(2,8.6)
(1,5.9)
(25,4.6)
(d.docid, $score_{BM25}(d, t_j)$)
descending $acc_d(t_i, t_j)$
PXL(bike,trails)
(2,3.0)
(4,0.7)
(12,0.5)
(9,0.2)
(d.docid, $acc_d(t_i, t_j)$)
CL(bike,trails)
(2,3.0,4.5,8.6)
(4,0.7,1.5,9.1)
(12,0.5,7.2,3.0)
(9,0.2,1.7,2.0)
(d.docid, $acc_d(t_i, t_j)$, $score_{BM25}(d, t_i)$, $score_{BM25}(d, t_j)$)

Figure 7.1: Score-ordered term, proximity, and combined index lists which can be used to process the query $\{bike, trails\}$ in several processing strategies.

The index structures depicted in Figure 7.1 can be combined into several processing strategies:

- `TL`: this corresponds to standard, text-based retrieval (just BM25 scores are employed) without usage of proximity scores. To process the query $\{bike, trails\}$, it uses the two term lists TL(bike) and TL(trails).

- `PXL`: this scans only the proximity lists and uses the proximity part of our modified Büttcher scoring function for ranking. To process the query $\{bike, trails\}$, this strategy uses the proximity list PXL(bike, trails).

- `TL+PXL`: this scans proximity and content score lists (which would be the straightforward implementation of our scoring model with a Threshold algorithm). To process the query {*bike, trails*}, this strategy uses the two term lists TL(bike) and TL(trails) as well as the proximity list PXL(bike, trails).

- `TL+CL`: this strategy, which is the main contribution of this chapter, exploits the additional content scores in the CLs to reduce the uncertainty about the score of documents with high proximity scores early in the process, which often allows early termination of the algorithm. We can additionally tighten the bounds when a CL for a pair (t_1, t_2) runs empty: if a document was seen in the TL for t_1, but not in the CL for (t_1, t_2), it is certain that it will not appear in the TL for t_2 any more. To process the query {*bike, trails*}, this strategy uses the two term lists TL(bike) and TL(trails) as well as the combined list CL(bike, trails).

We restrict ourselves to answering soft phrase queries. Once indexing considering term pair occurrences in a text window has been performed, it is not possible to process strict phrase queries with a pair-based index. This means that we cannot exclude those documents from the result set that do not contain the terms from the phrase consecutively. However, proximity scores are usually higher for documents with phrase occurrences than for those without phrase occurrences. Therefore, documents with phrase occurrences are not pruned away from the pair lists such that they will be very likely to be considered during query processing. Query-independent weights such as PageRank weights [BP98] may be stored in term lists. As they are small in size for commonly used document collections, they may be kept even in main memory (but this has also not been considered in other papers such as [DS11, SC07]). If updates are needed, the updates will just have to be carried out in one place - the term lists.

There has been a noticeable amount of work using precomputed lists for documents containing two or more terms to speed up processing of conjunctive queries, for example [CCKS07, KPSV09, LS05], for centralized search engines, and [PRL$^+$07] for distributed search engines. None of these approaches includes proximity scores, so they can only improve processing performance, not result quality. Another bunch of papers deals with efficiently precomputing indexes for phrase queries [BWZ02, CP06, WZB04], but again they do not include proximity scores. Some of these consider the problem of reducing the index size while providing decent performance for most queries, usually by restricting to phrases or term pairs in frequently occurring queries.

7.3.2 Evaluation Setup

We evaluated our algorithms with the Java-based, open-source TopX search engine[1] [TSW05] which stores index lists in an Oracle database. Our experiments were run using the GOV2 collection with roughly 25 million documents, corresponding to about 426 GB of data (see Section 3.2.1 for more details). We evaluated our methods with the 100 Ad Hoc topics (topic numbers 701-800) from the 2004 and 2005 TREC Terabyte Track, Ad Hoc Tasks. The topic sets are listed in Tables B.1 and B.2. As we are focusing on top-k retrieval, we measured precision values at several cutoffs.

[1] `http://topx.sourceforge.net`

To evaluate efficiency, we measured the number of sequential (SA) and random (RA) accesses to the index lists and the number of bytes transferred from disk, assuming sizes of 8 bytes for scores and docids. As random accesses are usually much more expensive than sequential accesses, we additionally compute a byte-based abstract cost

$$Cost(\gamma) = \#bytes(SA) + \gamma \cdot \#bytes(RA)$$

for each run, based on the cost ratio $\gamma := c_R/c_S$ of random to sequential accesses; we used γ values of 100 and 1,000 to determine abstract costs.

We indexed the documents with the indexer included in the TopX system with stopword removal enabled and computed the pair lists needed for the queries with an additional tool. We ran the results with TopX configured in `RR-LAST` mode and a batch size of 5,000, i.e., round-robin sequential accesses in batches of 5,000 items to the index lists and postponing random accesses to the end.

7.3.3 Results

Table 7.1 shows our experimental results for top-10 retrieval with stemming enabled.

Configuration	P@10	#SA	#RA	#bytes(SA)	#bytes(RA)	Cost(100)	Cost(1,000)
TL	0.56	24,175,115	196,174	386,801,840	1,569,392	543,741,040	1,956,193,840
TL+PXL	0.60	24,743,914	149,166	395,902,624	1,193,328	515,235,424	1,589,230,624
TL+CL	0.60	4,362,509	8,663	108,743,568	79,256	116,669,168	187,999,568
PXL	0.40	867,095	2,925	13,873,520	23,400	16,213,520	37,273,520

Table 7.1: Experimental results for top-10 retrieval of 100 Ad Hoc topics from the 2004 and 2005 TREC Terabyte Track, Ad Hoc Tasks.

It is evident that the configuration `TL+CL` improves P@10 to 0.60 over the original BM25 setting (which corresponds to `TL` with a P@10 value of 0.56), with a t-test and a Wilcoxon signed-rank test confirming statistically significant improvements at $p < 0.01$. The configuration `TL+PXL` with simple proximity lists achieves the same improvement in precision as it uses the same scoring function as `TL+CL` whereas scanning only the PXLs exhibits poor result precision. We verified by additional experiments that the retrieval quality of our modification of Büttcher's scoring model was as good as the original version of Büttcher's scoring model.

In addition to the improved retrieval quality of `TL+CL` over the `TL` baseline, it dramatically reduces the number of accesses, bytes transferred, and abstract costs by a factor of 5 to 10. This is due to the additional content scores available in `CL` and the better bounds. The configuration `TL+PXL` needs to run longer than `TL+CL` until it can safely stop. Scanning only the PXLs is much faster (at the expense of result quality).

index/limit	unpruned size(#items)	required space
TL	$3.191 \cdot 10^9$	47.5GB
PXL/CL (estimated!)	$1.410 \cdot 10^{12}$	20.5TB / 41.0TB

Table 7.2: Index sizes in items and required space for unpruned indexes.

Table 7.2 shows the index sizes (number of list entries and required space) for term (exact) and pair lists (estimated). As the complete set of pair lists was too large to completely materialize it, we randomly sampled 1,500,000 term pairs with a frequency of at least 10, of which about 1.2% had a non-empty pair list. They are calculated/estimated according to the kind of data stored in the lists as described in Section 7.3.1, assuming an uncompressed storage. We assume that document identifiers and scores have a size of 8 bytes each. Therefore one TL entry or PXL entry (consisting of document identifier and BM25 score or accumulated score, respectively) takes a size of 16 bytes whereas one CL entry takes a size of 32 bytes as it stores the document identifier, the accumulated score, and two BM25 scores.

It is evident that keeping all pair lists consumes prohibitively much disk space (for the GOV2 collection the estimated disk space to store unpruned PXL and CL indexes amounts to 20.5TB and 41.0TB, respectively): for large collections, the size of the inverted lists may be too large to completely store them, especially when the index includes term pair lists. As we do not consider only adjacent terms, but *any* terms occurring in the same document, a complete set of pair lists will be much larger than the original text collection.

Lossless index compression techniques (see, e.g., [dMNZBY00]) are one way to solve this problem, but the compression ratio will not be sufficient for really huge collections. We therefore apply *index pruning* (which is a lossy index compression technique) to reduce the size of the index, while at the same time sacrificing as little result quality as possible. Following the literature on inverted lists for text processing, a common way is pruning lists *horizontally*, i.e., dropping entries towards the end of the lists. These entries have low scores and hence will not play a big role when retrieving the best results for queries. Unlike term lists, term pair lists contain many entries with very low scores (as the score depends on the distance of term occurrences), so the pruning effect on pair lists should be a lot higher than on term lists.

7.3.4 Results with Pruned Index Lists

Our indexing framework provides three different pruning methods, mainly geared towards term pair lists. First, we heuristically limit the distance of term occurrences within a document, as occurrences within a large distance have only a marginal contribution to the proximity score. Second, we heuristically limit the list size to a constant, usually in the order of a few thousand entries. Third, we leverage the seminal work by Soffer et al. [SCC+01] for pair lists. They introduced list pruning with quality guarantees for the scores of query results, assuming top-k style queries with a fixed (or at least bounded) k. For each list L_i, they consider the score $s_k(L_i)$ at position k of the list, and drop each entry from that list whose score is below $\epsilon \cdot s_k(L_i)$, where $0 < \epsilon < 1$ is a tuning parameter.

We first study the size of our indexes at different levels of pruning for an index (without stemming as this is an upper bound for the index size with stemming). Table 7.3 shows the influence of index list pruning on the number of index items. It is

index/limit	500	1,000	1,500	2,000	2,500	3,000	unpruned
TL	295	355	402	442	472	496	3,191
PXL/CL (est.)	368,761	435,326	481,949	515,079	542,611	566,277	1,410,238
PXL/CL, $acc_d \geq 0.01$ (est.)	23,050	28,855	34,023	38,985	42,085	45,186	87,049

Table 7.3: Index sizes (million items) with different length limits, with and without minimum *acc*-score requirement.

index/limit	500	1,000	1,500	2,000	2,500	3,000	unpruned
TL	4.4GB	5.3GB	6.0GB	6.6GB	7.0GB	7.4GB	47.5GB
PXL (est.)	5.4TB	6.3TB	7.0TB	7.5TB	7.9TB	8.2TB	20.5TB
PXL, $acc_d \geq 0.01$ (est.)	343.5GB	430GB	507GB	580.9GB	627.1GB	673.3GB	1.3TB
CL (est.)	10.7TB	12.7TB	14.0TB	15.0TB	15.8TB	16.5TB	41.0TB
CL, $acc_d \geq 0.01$ (est.)	686.9GB	860GB	1.0TB	1.1TB	1.2TB	1.3TB	2.5TB

Table 7.4: Index sizes (disk space) with different length limits, with and without minimum *acc*-score requirement.

evident that keeping all pair lists, even with a length limit, is infeasible.

However, limiting the text window size to 10 reduces the number of items in the CL index noticeably to at most a factor of 8-15 over the unpruned term index, which may be tolerated given the cheap disk space available today. We mark settings with limited window sizes by $acc_d \geq 0.01$; one term occurrence of both t_i and t_j in a text window of 10 amounts to an $acc_d(t_i, t_j)$ contribution of at least 0.01.

Table 7.4 shows the index sizes (required disk space) for the very same lists. The size of TLs is not a big issue as the unpruned TLs only amount to 47.5GB, and can be further downsized using maximum list lengths. The far more critical indexes are PXLs and CLs that exhibit the prohibitive estimated size of 20.5TB and 41.0TB, respectively. Limiting the list size helps, although the lists remain too large. Additionally restricting PXLs and CLs by a minimum *acc*-score of 0.01 finally leads to tolerable sizes between 343.5GB and 673.3GB for PXLs and 686.9GB and 1.3TB for CLs.

As we show later in Table 7.5, excellent results can be achieved when limiting the index size to 1,000 entries per list. Hence, we need less than 900GB of disk space to execute `TL+CL(1,000;`$acc_d \geq 0.01$`)` on a document collection with 426GB data. Note that in this setting, both TLs and CLs keep at most 1,000 entries and CL entries require a minimum *acc*-score of 0.01. Additional lossless compression may further reduce the index sizes.

We then evaluated retrieval quality with pruned (term and combined) index lists, where we used combinations of window-based pruning with a maximal size of 10, fixed-length index lists, and the pruning technique by Soffer et al. [SCC$^+$01] for $k = 10$. All measurements were done without random accesses (i.e., using NRA), hence we report only a single cost value based on the number of bytes transferred by sequential accesses. Additional experiments without this constraint in `RR-LAST` mode showed that TopX only rarely attempts to make RAs in this setting as the pruned lists are often very short: hence, `RR-LAST` degenerates into NRA.

Table 7.5 shows the experimental results for top-10 queries in this setup, again with

Configuration	P@10	#SA	bytes(SA)	cost
TL+CL ($acc_d \geq 0.01$)	0.60	5,268,727	111,119,408	111,119,408
TL (500)	0.27	148,332	2,373,312	2,373,312
TL (1,000)	0.30	294,402	4,710,432	4,710,432
TL (1,500)	0.32	439,470	7,031,520	7,031,520
TL (2,000)	0.34	581,488	9,303,808	9,303,808
TL (2,500)	0.36	721,208	11,539,328	11,539,328
TL (3,000)	0.37	850,708	13,611,328	13,611,328
TL+CL (500)	0.53	295,933	7,178,960	7,178,960
TL+CL (1,000)	0.58	591,402	14,387,904	14,387,904
TL+CL (1,500)	0.58	847,730	20,605,312	20,605,312
TL+CL (2,000)	0.60	1,065,913	25,971,904	25,971,904
TL+CL (2,500)	0.60	1,253,681	30,648,064	30,648,064
TL+CL (3,000)	0.60	1,424,363	34,904,576	34,904,576
TL+CL ($\epsilon = 0.010$)	0.60	4,498,890	87,877,520	87,877,520
TL+CL ($\epsilon = 0.025$)	0.60	3,984,801	73,744,304	73,744,304
TL+CL ($\epsilon = 0.050$)	0.60	4,337,853	75,312,336	75,312,336
TL+CL ($\epsilon = 0.100$)	0.60	5,103,970	84,484,976	84,484,976
TL+CL ($\epsilon = 0.200$)	0.58	6,529,397	105,584,992	105,584,992
TL+CL (500; $\epsilon = 0.025$)	0.54	281,305	6,628,528	6,628,528
TL+CL (1,000; $\epsilon = 0.025$)	0.58	521,519	12,034,320	12,034,320
TL+CL (1,500; $\epsilon = 0.025$)	0.59	732,919	16,606,064	16,606,064
TL+CL (2,000; $\epsilon = 0.025$)	0.60	910,721	20,377,904	20,377,904
TL+CL (2,500; $\epsilon = 0.025$)	0.60	1,060,994	23,519,296	23,519,296
TL+CL (3,000; $\epsilon = 0.025$)	0.60	1,191,956	26,211,376	26,211,376
TL+CL (500; $acc_d \geq 0.01$)	0.58	290,788	6,931,904	6,931,904
TL+CL (1,000; $acc_d \geq 0.01$)	0.60	543,805	12,763,376	12,763,376
TL+CL (1,500; $acc_d \geq 0.01$)	0.61	780,157	18,117,552	18,117,552
TL+CL (2,000; $acc_d \geq 0.01$)	0.61	984,182	22,734,544	22,734,544
TL+CL (2,500; $acc_d \geq 0.01$)	0.61	1,166,144	26,854,608	26,854,608
TL+CL (3,000; $acc_d \geq 0.01$)	0.61	1,325,250	30,466,512	30,466,512

Table 7.5: Experimental results for top-10 retrieval with pruned lists.

stemming enabled. It is evident that `TL+CL` with length-limited lists and a minimum *acc*-score constraint (limited window size) gives a factor of 50-150 over the unpruned `TL` baseline in terms of saved cost, while yielding the same result quality (`TL+CL (1,000;` $acc_d \geq 0.01$`)`). Using `TL` as processing strategy with term lists of limited length is a lot worse in effectiveness. Pruning with ϵ is not as efficient, and large values for ϵ in fact *increase* cost: many entries from the pair lists are pruned away, but at the same time the additional content scores available from these entries are not available any more.

In combination with length limiting, results are comparable to our best configuration, but with slightly longer lists. Figures 7.2 to 7.5 illustrate some of these experimental results. We obtain the best precision values when limiting the list size to 1,500 or more elements (for `TL+CL(#items;` $acc_d \geq 0.01$`)` runs). Out of the approaches depicted in Figures 7.2 and 7.3, `TL+CL(#items)` is the approach with the worst precision values at the highest cost. `TL+CL(#items;` $acc_d \geq 0.01$`)` provides the best precision values at a medium cost, whereas `TL+CL(#items;` $\epsilon = 0.025$`)` only comes up with a slightly better precision than `TL+CL(#items)`, however at the best costs. For mere static index list pruning, precision values are most favorable for choices of ϵ below 0.1.

Table 7.6 demonstrates that, compared to `TL+CL` with pruned lists, `TL+PXL` with

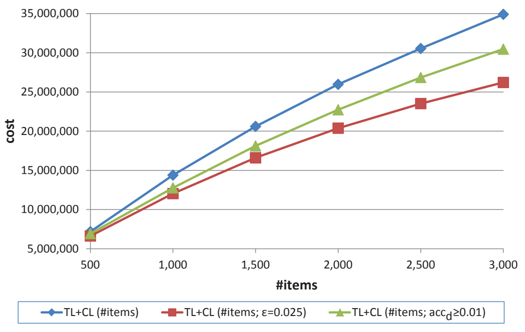


Figure 7.2: TL+CL approaches: cost.

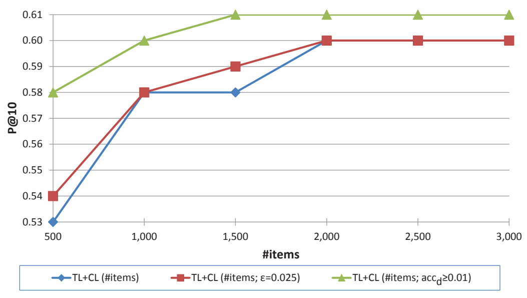


Figure 7.3: TL+CL approaches: P@10.

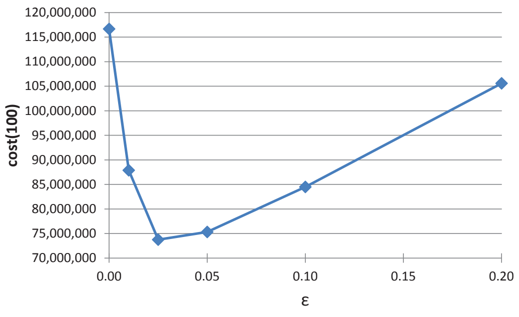


Figure 7.4: TL+CL(ϵ varied): cost.

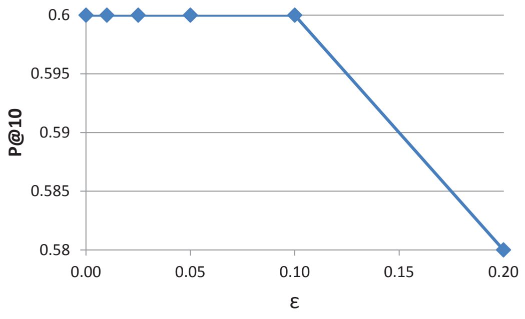


Figure 7.5: TL+CL(ϵ varied): P@10.

pruned lists suffer from a strongly reduced retrieval quality. This is due to the fact that documents from pruned CLs are often not among the top-documents in TLs such that their BM25 scores are missing in pruned TLs: these additional BM25 scores from the CLs have a decisive impact on the retrieval quality as the results of runs using `TL+PXL` with pruned lists deteriorate. Runs with minimum *acc*-score constraints and the pruning technique by Soffer et al. [SCC$^+$01] delivered comparable results. Hence, we do not consider `TL+PXL` settings with pruned lists any more.

Given that obvious importance of BM25 scores in pruned CLs for the high retrieval quality of `TL+CL(#items)`, we now investigate to which extent the result quality changes if we use only the BM25 scores from the pruned CLs.

To this end, we devise another list structure called CTL (combined term index list) that is based on CL but removes the *acc*-score dimension. For each single term pair (t_1, t_2) there is a list that contains an entry for each document d where this term pair occurs within any text window of size W. This entry has the form $(d.docid, score_{\text{BM25}}(d, t_1), score_{\text{BM25}}(d, t_2))$. t_1 and t_2 are lexicographically ordered (i.e., $t_1 < t_2$) such that for any single term pair combination we keep the corresponding combined list just once. CTL(t_1, t_2) denotes the combined term index list for the term pair (t_1, t_2). Pruned CTLs keep those entries from the corresponding CLs where the total contribution from both BM25 score dimensions is highest.

Table 7.7 compares the retrieval quality for top-100 retrieval of pruned `TL+CTL` and `TL+CL` settings. For a list length of 1,000, P@100 for `TL+CL` is comparable to `TL+CTL`;

Configuration	P@10	P@100
TL+CL ($acc_d \geq 0.01$)	0.60	0.39
TL+PXL (1,000)	0.43	0.26
TL+PXL (2,000)	0.45	0.28
TL+PXL (3,000)	0.47	0.30
TL+PXL (3,000; $\epsilon = 0.025$)	0.47	0.30
TL+PXL (3,000; $acc_d \geq 0.01$)	0.47	0.30
TL+CL (1,000)	0.58	0.37
TL+CL (2,000)	0.60	0.38
TL+CL (3,000)	0.60	0.39

Table 7.6: Retrieval quality for top-10 and top-100 retrieval with pruned lists.

Configuration	P@100
TL+CTL(1,000)	0.3768
TL+CTL(2,000)	0.3753
TL+CTL(3,000)	0.3769
TL+CL (1,000)	0.3710
TL+CL (2,000)	0.3841
TL+CL (3,000)	0.3877

Table 7.7: Retrieval quality for top-100 retrieval with pruned TL+CTL and TL+CL settings.

for longer lists, `TL+CL` outperforms `TL+CTL`. Due to this advantage, in Chapter 8, we will focus on `TL+CL` settings when we determine pruning levels for index structures.

Podnar et al. [PRL+07] use static index list pruning in a peer-to-peer setting with terms and term sets as keys. They distinguish discriminative keys (DKs) that occur in at most DF_{max} documents and non-discriminative keys (NDKs) that occur in more than DF_{max} documents. Posting lists of NDKs are truncated to their best DF_{max} entries. A key is called intrinsically discriminative if it is a DK and all smaller subsets of this key are NDKs. A key is called highly discriminative key (HDK) if it has at most s_{max} terms, and it occurs in a window of size w, and in addition is an intrinsically discriminative key. Full posting lists are stored for HDKs. Each peer maintains a local index which is built in several iterations, starting with one-term keys to s_{max}-term keys, and adds local HDKs and NDKs with their posting lists to the global network. The P2P network maintains the global posting lists and notifies the responsible peers if an inserted HDK becomes globally non-discriminative. In that case, the peers in charge of the globally non-discriminative keys start expanding the keys with additional terms to produce new HDKs of increased key size. The authors use BM25 scores as a scoring model which is similar to using `TL+CTL`(DF_{max}).

As especially pruning along the lines of Soffer et al. [SCC+01] is done for a specific value of k, it is interesting to see how good results using the index pruned with $k = 10$ are for larger values of k. Tables 7.8 and 7.9 show the results for top-100 retrieval with pruned and unpruned lists. Even though proximity awareness cannot improve much on result quality, most runs with pruning are at least as effective as the unpruned runs, while saving one or two orders of magnitude in accesses, bytes transferred, and cost. The combination of length-limited lists and limited window size is again best, with a peak factor of more than 320 over the unpruned `TL` baseline at the same quality (`TL+CL (1,000;` $acc_d \geq 0.01$`)`).

Configuration	P@100	MAP@100	#SA	#RA	#bytes(SA)	#bytes(RA)
TL	0.37	0.13	42,584,605	434,233	681,353,680	3,473,864
TL+PXL	0.39	0.14	44,450,513	394,498	711,208,208	3,155,984
TL+CL	0.39	0.14	12,175,316	32,357	302,386,896	380,552
PXL	0.27	0.09	867,095	2,925	13,873,520	23,400
TL+CL ($acc_d \geq 0.01$)	0.39	0.14	17,714,952	0	346,997,712	0
TL+CL (500)	0.34	0.11	310,469	0	7,558,816	0
TL+CL (1,000)	0.37	0.13	610,983	0	14,838,144	0
TL+CL (1,500)	0.38	0.13	904,910	0	21,911,520	0
TL+CL (2,000)	0.38	0.14	1,184,658	0	28,615,776	0
TL+CL (2,500)	0.39	0.14	1,457,093	0	35,138,176	0
TL+CL (3,000)	0.39	0.14	1,723,204	0	41,493,728	0
TL+CL (500; $\epsilon = 0.025$)	0.33	0.11	281,485	0	6,631,408	0
TL+CL (1,000; $\epsilon = 0.025$)	0.36	0.12	527,171	0	12,156,256	0
TL+CL (1,500; $\epsilon = 0.025$)	0.37	0.13	753,012	0	17,054,112	0
TL+CL (2,000; $\epsilon = 0.025$)	0.37	0.13	957,593	0	21,371,376	0
TL+CL (500; $acc_d \geq 0.01$)	0.34	0.12	290,968	0	6,934,784	0
TL+CL (1,000; $acc_d \geq 0.01$)	0.37	0.13	551,684	0	12,940,576	0
TL+CL (1,500; $acc_d \geq 0.01$)	0.38	0.13	802,538	0	18,638,752	0
TL+CL (2,000; $acc_d \geq 0.01$)	0.38	0.13	1,039,466	0	23,969,632	0
TL+CL (2,500; $acc_d \geq 0.01$)	0.38	0.13	1,261,124	0	28,907,200	0
TL+CL (3,000; $acc_d \geq 0.01$)	0.38	0.13	1,483,154	0	33,856,144	0

Table 7.8: Experimental results for top-100 retrieval with unpruned and pruned lists.

Configuration	Cost(100)	Cost(1,000)
TL	1,028,740,080	4,155,217,680
TL+PXL	1,026,806,608	3,867,192,208
TL+CL	340,442,096	682,938,896
PXL	16,213,520	37,273,520
TL+CL ($acc_d \geq 0.01$)	346,997,712	346,997,712
TL+CL (500)	7,558,816	7,558,816
TL+CL (1,000)	14,838,144	14,838,144
TL+CL (1,500)	21,911,520	21,911,520
TL+CL (2,000)	28,615,776	28,615,776
TL+CL (2,500)	35,138,176	35,138,176
TL+CL (3,000)	41,493,728	41,493,728
TL+CL (500; $\epsilon = 0.025$)	6,631,408	6,631,408
TL+CL (1,000; $\epsilon = 0.025$)	12,156,256	12,156,256
TL+CL (1,500; $\epsilon = 0.025$)	17,054,112	17,054,112
TL+CL (2,000; $\epsilon = 0.025$)	21,371,376	21,371,377
TL+CL (2,500; $\epsilon = 0.025$)	25,288,646	25,288,646
TL+CL (3,000; $\epsilon = 0.025$)	28,924,720	26,211,376
TL+CL (500; $acc_d \geq 0.01$)	6,934,784	6,934,784
TL+CL (1,000; $acc_d \geq 0.01$)	12,940,576	12,940,576
TL+CL (1,500; $acc_d \geq 0.01$)	18,638,752	18,638,752
TL+CL (2,000; $acc_d \geq 0.01$)	23,969,632	23,969,632
TL+CL (2,500; $acc_d \geq 0.01$)	28,907,200	28,907,200
TL+CL (3,000; $acc_d \geq 0.01$)	33,856,144	33,856,144

Table 7.9: Costs for top-100 retrieval with unpruned and pruned lists.

7.3.5 Comparison: TopX(`RR-LAST` Mode) on Unpruned Lists vs. Merge Join on Pruned Lists

Our experiments so far have shown that lists cut at a maximum length of 2,000 (when combined with window- or epsilon-based pruning even less) can retain the retrieval quality of unpruned index lists. So we might be able to save the overhead costs induced by the family of threshold algorithms.

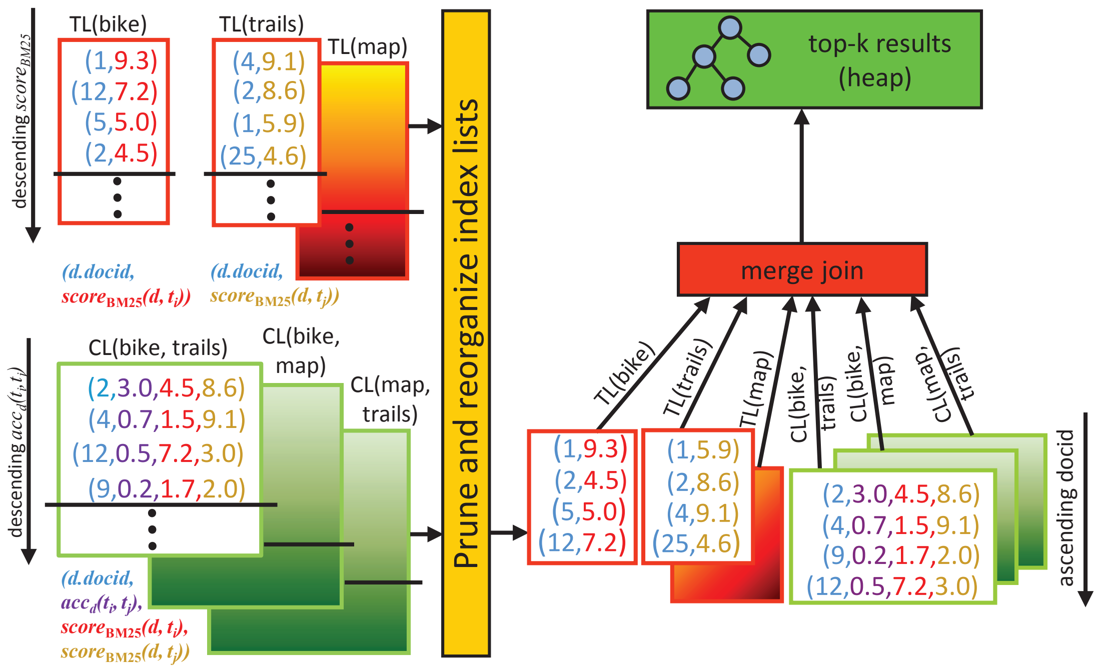


Figure 7.6: Example: query={*bike, trails, map*}, merge join with processing strategy TL+CL using pruned term lists and combined lists.

The highly efficient *top-k* or *dynamic pruning* algorithms [AM06, FLN03] that are frequently applied for efficient query processing incur a non-negligible processing overhead for maintaining candidate lists and candidate score bounds, for mapping newly read index entries to a possibly existing partially read document using hash joins, and for regularly checking if the algorithm can stop. In our scenario with index lists that are pruned to a rather short maximal length, this processing overhead is not necessary since we will almost always read the complete lists anyway. Instead, it is sufficient to evaluate queries in document-at-a-time evaluation. Our merge-based processing architecture is depicted in Figure 7.6 for index lists relating to the example query terms *bike*, *trails*, and *map*, and consists of the following components:

1. After pruning index lists to a fixed maximal size (and, possibly, using a minimal score cutoff for combined lists), we resort each list in ascending order of docids, and optionally compress it.

2. At query time, the n term and combined lists for the query are combined using an n-way merge join that combines entries for the same document and computes its score. The n-way merge join receives the n document-sorted lists as input and in each join step calculates the score for the next smallest not yet evaluated docid. If that score is higher than the current kth best score, the document is kept in a heap of candidate results (e.g., described on page 125 in [MRS08]), otherwise it is dropped as it cannot make it into the top-k results any more. For every list it keeps track of the position up to which the list has been read so far and iterates to the next item if the document in this list has just been evaluated. If the items of all lists have been read completely, the algorithm terminates. Note that we process queries in a disjunctive manner, i.e., docids that do not occur in every list can still qualify for the top-k results.

3. Once all index entries have been read, the content of the heap is returned.

Instead of maintaining a heap with the currently best k results, an even simpler implementation could keep all results as result candidates and sort them in the end; however, this would increase the memory footprint of the execution as not k, but all encountered documents and their scores need to be stored.

Independent of the actual algorithm, processing a query with our pruned index lists has a guaranteed maximal abstract execution cost (i.e., the number of index entries read from disk during processing a query), so worst- and best-case runtime are very similar and basically depend only on the number of lists involved in the execution and the cutoff for list lengths. This is a great advantage over using non-pruned term lists with algorithms for dynamic pruning and early stopping, which can read large and uncontrollable fractions of the index lists to compute the results, and may give arbitrarily bad results when stopped earlier [SSLM$^+$09].

Our experiments use a server running the Microsoft Windows 2003 Enterprise 64-bit edition on a Dual Core AMD Opteron CPU with 2.6GHz and 32 GB RAM. Both TopX and the merge join-based approach have been executed in a Sun Java 1.6 VM that was allowed to use at maximum 4 GB RAM, although the real memory requirements are way below 4 GB. Index lists have been stored in an Oracle 10g DBMS. The baseline of the experiments has been computed with TopX with a batch size of 100,000 in `RR-LAST` mode, i.e., round-robin sequential accesses to the index lists in batches of 100,000 items and postponing the random accesses to the end of the query processing. Unlike the previous experiments where we used only abstract cost measures, now, we will show that the abstract cost advantages translate into accelerated query processing by measuring real runtimes (in ms).

Table 7.10 shows average precision values and measured runtimes (in ms) for various TopX- and merge join-based runs. Please note that we encounter minor differences in result quality compared to the experiments carried out in the first part of this chapter. This is due to a modified policy for tie-breaking; in Table 7.10 documents with lower docids are given preference in case of equally scored documents. The order of result quality and execution speed are still the same, however.

		k=10		k=30		k=50		k=70		k=100	
	Run	P@k	t[ms]	P@k	t[ms]	P@k	t[ms]	P@k	t[ms]	P@k	t[ms]
TopX	TL	0.57	5,220	0.49	6,129	0.45	6,868	0.42	7,974	0.38	8,827
	TL+PXL	0.61	6,266	0.52	8,835	0.48	11,127	0.45	10,363	0.41	15,524
	TL+CL	0.61	821	0.52	1,350	0.48	1,651	0.45	1,955	0.41	2,042
Merge Join	TL (1,000)	0.30	35	0.24	36	0.22	35	0.20	34	0.18	35
	TL (2,000)	0.34	68	0.27	71	0.24	64	0.22	67	0.20	67
	TL (3,000)	0.37	101	0.29	103	0.27	100	0.25	99	0.22	100
	TL+CL (1,000)	0.60	78	0.51	80	0.46	80	0.43	81	0.39	79
	TL+CL (2,000)	0.61	156	0.51	155	0.47	153	0.44	155	0.40	158
	TL+CL (3,000)	0.61	225	0.52	230	0.47	224	0.44	223	0.40	232
	TL+CL (1,000; $\epsilon = 0.025$)	0.60	79	0.51	80	0.46	80	0.43	81	0.39	81
	TL+CL (2,000; $\epsilon = 0.025$)	0.61	154	0.51	153	0.47	153	0.44	155	0.40	153
	TL+CL (3,000; $\epsilon = 0.025$)	0.61	223	0.52	223	0.47	225	0.44	225	0.40	224
	TL+CL (1,000; $acc_d \geq 0.01$)	0.62	70	0.51	73	0.46	70	0.42	71	0.38	76
	TL+CL (2,000; $acc_d \geq 0.01$)	0.62	137	0.51	135	0.47	134	0.44	135	0.39	134
	TL+CL (3,000; $acc_d \geq 0.01$)	0.62	193	0.51	199	0.47	191	0.44	191	0.40	191

Table 7.10: Comparison: TopX with unpruned lists vs merge join on pruned lists.

TL+CL runs are faster than TL+PXL runs at similar precision values, and TL runs exhibit a decreased precision compared to TL+CL and TL+PXL runs. While run times for TopX runs usually grow with increasing k, run times of the merge join runs are independent of k: TopX can terminate the query evaluation early dependent on the number of results; for merge joins, the lists are read completely, no matter how k is chosen. For the merge join implementation, run times are linearly proportional to the lengths of the read lists.

The merge join implementation of the pruned TL+CL lists can keep up the excellent precision values of the TopX runs with unpruned lists. Using a light-weight n-ary merge join in combination with pruned index lists sorted by docid, we achieve substantial performance gains. That way, we save much disk space and accelerate query processing by one to two orders of magnitude compared to an evaluation with TopX using unpruned lists.

7.3.6 Conclusion of the Experiments

We have presented novel algorithms and implementation techniques for efficient evaluation of top-k queries on text data with proximity-aware scoring. We have shown that our techniques can speed up evaluation by one or two orders of magnitude, trading in runtime for cheap disk space and maintaining the very high result quality (effectiveness) of proximity-aware scoring models.

Furthermore, we have shown that the abstract cost advantages can be turned into substantial runtime benefits using a light-weight n-ary merge join in combination with pruned document-sorted index lists. The speed up by one or two orders of magnitude compared to an evaluation with TopX in `RR-LAST` mode using unpruned lists can be confirmed, still providing the same excellent precision values and in addition saving much disk space.

7.4 Feasibility of Scoring Models for Top-k Query Processing

While some of the techniques presented in Chapter 2 demonstrate significant improvements in result quality (cf. Chapter 4), they do not consider the problem how these scores can be efficiently implemented in a search engine. Usually, implementations therefore resort to enriching term index lists with position information (e.g., [YDS09a]) and compute proximity scores after having determined an initial set of documents with 'good' text scores (e.g., cf. Section 2.4.1).

Orthogonal to this kind of works, we discuss the feasibility of scoring models for top-k query processing with early termination in the sense of whether they can be used for NRA-based query processing (cf. Section 6.1). Early termination means that the index lists do not have to be processed completely but reading can stop usually long before all index entries have been read. In particular, we want to explore whether the scoring models are suitable for usage with the index model based on pair lists as presented in Section 7.3.1. Assessing this involves a judgment whether we can build a queryload-independent index that allows to precompute score bounds for result candidates during query processing (cf. Section 7.2.2) to enable early termination by means of tight score bounds.

7.4.1 Linear Combinations of Scoring Models

We start our discussion with the class of linear combinations of scoring models which combine content and proximity score models.

Rasolofo and Savoy: Rasolofo and Savoy process queries in two steps: step one computes the k documents with the highest *cscore* values, step two re-ranks the documents from step one using proximity scores. One way to make Rasolofo and Savoy's approach feasible for top-k query processing works in analogy to the approach used for the modification of Büttcher's score (Section 7.2.2) and employs term and term pair lists to store for *cscore* and *pscore* computations, respectively. Rasolofo and Savoy use

$$score_{\mathrm{BM25}}(d,q)=\sum_{t_i\in q}\frac{(k_1+1)\cdot tf(t_i,d)}{k\cdot[(1-b)+b\cdot\frac{l_d}{avgdl}]+tf(t_i,d)}\cdot max\{0,\log\frac{N-df(t_i)}{df(t_i)}\}\cdot qtf'(t_i),$$

where $qtf'(t_i)=\frac{qtf(t_i)}{k_3+qtf(t_i)}$. A term list for term t_i could keep an entry of the form $(d.docid, score_{\mathrm{BM25}}(d,t_i)/qtf'(t_i))$ for each document d with at least one occurrence of term t_i. The list is ordered by descending $score_{\mathrm{BM25}}(d,t_i)/qtf'(t_i)$ values. Dividing $score_{\mathrm{BM25}}(d,t_i)$ by $qtf'(t_i)$ removes the query dependency from $score_{\mathrm{BM25}}(d,t_i)$.

Additionally, the term list can maintain the $idf_2(t_i)$ score in order to compute $qw(t_i)=idf_2(t_i)\frac{qtf(t_i)}{k_3+qtf(t_i)}$ which is required to compute *pscore* values. A term pair list for a term pair (t_i,t_j) can be used for *pscore* computations if the text window size *dist* is kept fixed. For each document with at least one occurrence of the term pair in the text window, it contains an entry of the form $(d.docid, w_d(t_i, t_j, dist))$ and is sorted by descending $w_d(t_i, t_j, dist)$ values.

During query processing, the BM25 score-related part in the term list is multiplied by $qtf'(t_i)$, which, for a given query, remains constant for all entries of t_i's term list. The resulting *cscore* for t_i is combined with the *pscore* part which can be derived from $w_d(t_i, t_j, dist)$ and $qw(t_i), qw(t_j)$, respectively.

Top-k query processing should definitely be applied to step one to allow early termination. Step two could use either random accesses to fetch proximity scores just for the documents from step one if the term pair index supports random accesses, or step two could scan the complete term pair lists, skipping documents not retrieved in step one. In the latter case it may be conceivable to order term pair lists by docid (best with a block structure and skip pointers) and perform a merge join of the document-sorted result lists from step one with the term pair lists.

Optionally, query processing could merge the two steps into one and process term and term pair lists together in a single step. Considering *cscore* and *pscore* at the same time, this approach would not re-rank just the top-k documents with the highest BM25 scores. Thus, this approach would omit the precomputation of the top-k documents with the highest BM25 scores. Like our top-k variant of Büttcher's score (cf. Section 7.2.2), this could help faster termination through tighter earlier score bounds.

Uematsu et al.: Adapting Uematsu et al.'s approach for top-k query processing is not easily possible: when it comes to compute score bounds for the *pscore* part, we need to know the number of sentences where **all** query terms co-occur, $co_{occ}(q, d)$. Building up lists with the number of co-occurrences for all possible query term combinations would solve that problem. As the query load is not known in advance, this will render indexes quickly prohibitively large.

As a compromise, we may store *d.docid* and $co_{occ}(q, d)$ values in lists ordered by descending co_{occ} value just for two-term-queries or selected frequent queries from query logs. For the general case of n-term queries however, the precomputation of *pscore* values is problematic as indexes are likely to end up being too large.

To save on runtime, an approximation comparable to Tonellotto et al.'s approach (cf. Section 6.2.2) could be used: $co_{occ}(q, d)$ cannot be greater than the minimum of $co_{occ}(\{t_i, t_j\}, d)$ for all $\{t_i, t_j\} \subseteq q$ which could be used as an upper bound for $co_{occ}(q, d)$.

One could generate term pair lists for (t_i, t_j) with $(d.docid, co_{occ}(\{t_i, t_j\}, d))$ entries ordered by descending $co_{occ}(\{t_i, t_j\}, d)$ values. For each term t, we build a term list (ordered by descending BM25 score) with $(d.docid, score_{\mathrm{BM25}}(d, t))$ entries in the same way as for Büttcher's approach (with a modified *idf* score) but with additional sentence-level posting lists: it may be necessary to use position lists to clarify the final $co_{occ}(q, d)$ value since $co_{occ}(q, d)$ is not decomposable into $co_{occ}(\{t_i, t_j\}, d)$ values.

Monz: Monz' approach cannot be straight-forwardly incorporated into top-k query processing.

The Lnu.ltc score (*cscore*) could be computed by means of an lnu-score ordered term list for each term t. That term list keeps the $idf(t)$ value and a list of $(d.docid, lnu(d,t))$ tuples for each document that contains t. The $ltc(t)$ value (which uses only *idf* and qtf values) can be computed before reading the first entry from the lnu list and factorized into the documents' Lnu.ltc scores. However, normalization by the maximum *Lnu.ltc* score achievable by any document in the collection is an issue: it requires knowledge about the top-1 result over the complete query and cannot yet be computed after reading only the first entry of each term list.

Computing minimum matching spans is inherently query-dependent. Their computation requires knowledge about the positions of query terms and the subset of query terms that occur in each document. Hence, precomputing the *pscore* which builds upon minimum matching spans is problematic as we do not know the queryload in advance. For two-term queries, proximity scores (product of *ssr* and *mtr* features) can be stored in a term pair list. In principle, for n-term queries ($n > 2$) this is doable as well; however space requirements to store lists will quickly render this approach non-practical. Anyway, precomputation could be done for selected, very frequent queries from a query log.

Keeping posting lists with term position information for any (term, document)-pair would be sufficient to compute minimum matching spans but lacks support to precompute score bounds for the *pscore* part.

Tao and Zhai: Tao and Zhai combine a baseline content score, which can be either the KL-divergence or the BM25 score, with a proximity score.

The content score part can be easily stored in term lists which allow top-k query processing with dynamic pruning. A term list for term t keeps an entry for each document d with at least one occurrence of term t. Term lists that use KL-divergence as *cscore*, maintain entries of the form $(d.docid, score_{KL}(d,t)/qtf(t))$, where $score_{KL}(d,t)/qtf(t) = ln(1 + \frac{tf(t,d)}{\mu \cdot p(t|\mathcal{C})}) + ln\frac{\mu}{l_d+\mu}$ and $p(t|\mathcal{C}) = \frac{ctf(t)}{l_\mathcal{C}}$. Term lists that use BM25 scores as *cscore*, follow the schema described for Rasolofo and Savoy's approach and contain $(d.docid, score_{\text{BM25}}(d,t)/qtf'(t))$ entries. The lists are ordered by descending $score_{KL}(d,t)/qtf(t)$ and $score_{\text{BM25}}(d,t)/qtf'(t)$ values, respectively. The qtf and qtf' values are constant for a given query-term combination and are incorporated while scoring a document at query processing time.

Tao and Zhai propose a *pscore* of the general form $\pi(d,q) = log(\alpha + e^{-\delta(d,q)})$ that employs a selection of span-based and distance aggregation measures to populate $\delta(d,q)$.

Span-based measures require position information and are inherently query-dependent so that we cannot precompute scores. To compute the *Span* value, we need knowledge about the maximum and minimum position of all query term occurrences in a document. Building up a term list for each term with minimum and maximum position of that term's occurrences in a document is not good enough. As long as there is just one missing dimension, the maximum and minimum position can still change (unless they are already 1 and l_d, respectively); score bounds cannot be precomputed that way. Term pair lists can resolve this issue just for two-term queries and use the difference of maximum and minimum position. To compute the $MinCover$ value, we face a similar problem, since the length of the shortest document part that covers each query term at least once has to be found. It seems that $MinCover$ cannot be decomposed into term pair lists to perform early candidate pruning as position information is required for all query terms. The normalized versions of *Span* and $MinCover$ share the same problems.

The situation is better for distance aggregation measures which can be represented by a term pair list for each term pair $\{t_a, t_b\}$. For each document d_i with at least one occurrence of both t_i and t_j, the term pair list for $\{t_a, t_b\}$ contains an entry of the form $(d_i.docid, mindist(t_a, t_b, d_i))$, where $mindist(t_a, t_b, d_i) = min\{|a - b| : p_a(d_i) = t_a \wedge p_b(d_i) = t_b\}$. Term pair lists may be sorted either by descending or ascending score since we will see in the following descriptions, that we need both $high_j$ and low_j scores to compute score bounds which is different from typical scenarios involving the NRA where lists are ordered by descending score. Therefore, it may be good to read term pair lists from both ends of the lists in order to decrease their high and increase their low values at the same time. If the list is sorted by descending score, $high_j$ is the score at the current scan position and low_j the lowest score available in a list, respectively. If the list is sorted by ascending score, $high_j$ is the highest score available in a list and low_j the score at the current scan position in a list, respectively.

We reuse the notation from Section 6.1 to compute a document d_i's *bestscore* and $worstscore$ values (c and p indexes denote the *cscore* and *pscore* component, respectively):

$$bestscore(d_i) = \frac{1}{2} \cdot bestscore_c(d_i) + \frac{1}{2} \cdot bestscore_p(d_i) \text{ and}$$

$$worstscore(d_i) = \frac{1}{2} \cdot worstscore_c(d_i) + \frac{1}{2} \cdot worstscore_p(d_i).$$

For a given query, L_{cscore} is the set of term lists, L_{pscore} is the set of term pair lists, both with remaining unread entries. $S_c(d_i) \subseteq L_{cscore}$ and $S_p(d_i) \subseteq L_{pscore}$ denote the set of term and term pair lists where d_i has been seen, whereas $\bar{S}_c(d_i) = L_{cscore} - S_c(d_i)$ and $\bar{S}_p(d_i) = L_{pscore} - S_p(d_i)$ represent the set of not completely processed term and term pair lists where d_i has not yet been encountered.

The minimum pair distance (*MinDist*) is the smallest distance over all query term pairs in document d_i with $MinDist(d_i, q) = min_{t_a, t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b}\{mindist(t_a, t_b, d_i)\}$. Hence, $\pi(d_i, q) = log(\alpha + e^{-\delta(d_i,q)}) = log(\alpha + e^{-min_{t_a,t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b}\{mindist(t_a,t_b,d_i)\}})$.

The corresponding $worstscore$ and *bestscore* for the proximity part $\pi(d_i, q)$ with *MinDist* can be calculated as follows:

$$worstscore_p(d_i) = log(\alpha + e^{-min(max_{L_j \in \bar{S}_p(d_i)}(high_j), min_{L_j \in S_p(d_i)}(mindist(t_a,t_b,d_i)))}) \text{ and}$$

$$bestscore_p(d_i) = log(\alpha + e^{-min(min_{L_j \in \bar{S}_p(d_i)}(low_j), min_{L_j \in S_p(d_i)}(mindist(t_a,t_b,d_i)))}).$$

As $worstscore_p(d_i)$ represents the minimally possible proximity score for d_i, the exponent of the exponential function must become as negative as possible. Consequently, we use the maximum among all $high_j$ for $L_j \in \bar{S}_p(d_i)$ as higher values render the argument of $-min$ larger and the exponent more negative. As $bestscore_p(d_i)$ represents the maximally possible proximity score for d_i, the exponent of the exponential function must become as little negative as possible. Hence, we use the minimum of all low_j values as lower values render the argument of $-min$ smaller and the exponent less negative. $min_{L_j \in S_p(d_i)}(mindist(t_a, t_b, d_i))$ represents $MinDist$ in the dimensions where d_i has already been encountered and is used both for $worstscore$ and $bestscore$ computations. The exponent starts with $-min$ as $-\delta(d_i, q) = -MinDist(d_i, q)$.

The average pair distance (*AvgDist*) is the average distance over all query term pairs in document d_i with $AvgDist(d_i, q) = \frac{2}{n(n-1)} \sum_{t_a, t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b} mindist(t_a, t_b, d_i)$ and n being the number of unique matched query terms in d_i. Hence,

$$\pi(d_i, q) = log(\alpha + e^{-\delta(d_i,q)}) = log(\alpha + e^{\frac{2}{n(n-1)} \sum_{t_a, t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b} mindist(t_a, t_b, d_i)}).$$

$\frac{2}{n(n-1)}$ corresponds to $\frac{1}{\binom{n}{2}}$ which is the reciprocal of the number of term pairs and used to build the average distance over all query term pairs.

The *worstscore* and *bestscore* for the proximity part $\pi(d_i, q)$ with *AvgDist* can be calculated as follows:

$$worstscore_p(d_i) = \log(\alpha + e^{-max_{A \subseteq \bar{S}_p(d_i)}[f(A, S_p(d_i))(\sum_{L_j \in A} high_j + \sum_{L_j \in S_p(d_i)} mindist(t_a, t_b, d_i))]})$$

and $bestscore_p(d_i) = \log(\alpha + e^{-min_{A \subseteq \bar{S}_p(d_i)}[f(A, S_p(d_i))(\sum_{L_j \in A} low_j + \sum_{L_j \in S(d_i)} mindist(t_a, t_b, d_i))]})$,

where $f(A, S_p(d_i)) = \frac{2}{(|A| + |S_p(d_i)|)(|A| + |S_p(d_i)| - 1)}$ and $(|A| + |S_p(d_i)|)$ represents the true value for n in d_i.

As $worstscore_p(d_i)$ represents the minimally possible proximity score for d_i, we have to select the subset A of term pair lists where d_i has not been encountered yet (i.e., $\bar{S}_p(d_i)$) such that the exponent of the exponential function becomes as negative as possible. We use $high_j$ for $L_j \in A$ as higher values render the argument of $-max$ larger and the exponent more negative. In contrast, $bestscore_p(d_i)$ represents the maximally possible proximity score for d_i: we have to select the subset A of term pair lists where d_i has not been encountered yet (i.e., $\bar{S}_p(d_i)$) such that the exponent of the exponential function is as little negative as possible. We use low_j for $L_j \in A$ as lower values render the argument of $-min$ smaller and the exponent less negative. Both for $worstscore_p(d_i)$ and $bestscore_p(d_i)$, $\sum_{L_j \in S_p(d_i)} mindist(t_a, t_b, d_i)$ represents the contribution of proximity score dimensions, where d_i has already been seen.

The maximum pair distance (*MaxDist*) is the maximum distance over all query term pairs in document d_i with $MaxDist(d_i, q) = max_{t_a, t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b}\{mindist(t_a, t_b, d_i)\}$. Hence, $\pi(d_i, q) = log(\alpha + e^{-\delta(d_i,q)}) = log(\alpha + e^{max_{t_a, t_b \in T_{d_i}(P_{d_i}(q)), t_a \neq t_b}\{mindist(t_a, t_b, d_i)\}})$.

The *worstscore* and *bestscore* for the proximity part $\pi(d_i, q)$ using *MaxDist* can be calculated as follows:

$$worstscore_p(d_i) = log(\alpha + e^{-max(max_{L_j \in \bar{S}_p(d_i)}(high_j), max_{L_j \in S_p(d_i)}(mindist(t_a, t_b, d_i)))}) \text{ and}$$

$$bestscore_p(d_i) = log(\alpha + e^{-max(min_{L_j \in \bar{S}_p(d_i)}(low_j), max_{L_j \in S_p(d_i)}(mindist(t_a, t_b, d_i)))}).$$

While $S_p(d_i) = \emptyset$, $worstscore_p(d_i) = \log(\alpha + e^{-l_{d_i}})$ for all distance aggregation measures. This takes account of the case that there is only one query term match in a document, for which $MinDist$, $MaxDist$ and $AvgDist$ are all defined as the length of the document l_{d_i}.

To safely stop, the following inequality must be fulfilled for not yet seen (virtual) documents:

$$\frac{1}{2} \cdot \sum_{L_j \in L_{cscore}} qtf'(t_j) high_j + \frac{1}{2} \cdot \log(\alpha + e^{-min_{L_j \in L_{pscore}}(low_j)}) < min\text{-}k.$$

The left side of the inequation represents the *bestscore* for not yet seen documents, the right side the smallest worstscore of the temporary top-k results. This holds for all presented distance aggregation measures. For $AvgDist$ and $MaxDist$, $bestscore_p$ becomes the largest if the virtual document is contained only in one term pair list, the one with the lowest low_j score. For $MaxDist$, we can safely remove the maximum from the exponent then (as there is only one dimension). For $MinDist$, $bestscore_p$ becomes the largest if the virtual document is contained in the term pair list with the lowest low_j score. The only difference to $AvgDist$ and $MaxDist$ is that it may also occur in other term pair lists; however this would not affect the $bestscore_p$ bound of the virtual document.

7.4.2 Integrated Score Models

This subsection discusses the feasibility of selected integrated score models for top-k query processing.

De Kretser and Moffat: De Kretser and Moffat describe two algorithms to compute a ranking of documents. As the first algorithm follows a greedy approach, we focus on the second. The second algorithm uses the maximum score among the scores at positions of query term occurrences in d as document score. The algorithm could be implemented by means of lists for ordered term pairs (t_i, t_j), $PL(t_i, t_j)$. They consist of entries of the form $(d.docid, x, \sum_{l \in P_d(t_j)} c'_{t_j}(x, l))$ ordered by descending $\sum_{l \in P_d(t_j)} c'_{t_j}(x, l)$ values and x being a position with a term occurrence of term t_i, i.e., $x \in P_d(t_i)$. $c'_{t_j}(x, l)$ equals $c_{t_j}(x, l)/qtf(t_j)$ as $qtf(t_j)$ values from the height component in $c_{t_j}(x, l)$ are only known at query processing time. Evaluating a query q processes all $PL(t_i, t_j)$ indexes, where $t_i, t_j \in q$. In this setup, $bestscore(d, q)$ equals $max_{x \in P_d(q)} bestscore(d, q, x)$, where $bestscore(d, q, x)$ is the highest score achievable at a query term position x. Note that this approach may not be usable in practice due to blown-up indexes: every document generates not only one entry per term pair (t_i, t_j), but one entry per occurrence of t_i in d. In addition, as the term pairs are ordered, we need two term pair lists per term pair.

Song et al.: Song et al. partition documents into groups of subsequent query term occurrences, so-called espans. Espans (the number of query terms in them and their density, respectively) do not seem to be representable by term or term pair lists as they highly depend on the position of all query term occurrences in a document: the solution to the problem of determining *bestscore* bounds for documents could follow an approach similar to the approach we sketched in Section 7.2.2 for Büttcher et al.'s score. Again, everything boils down to a combinatorial problem where we construct conceived documents by means of $tf(t,d)$ scores that maximize the document score which is dependent on spans. Compared to the solution for Büttcher et al.'s score, this problem seems to be even more difficult as the document has to be additionally split into espans.

Mishne and de Rijke: Mishne and de Rijke's scoring model follows an "everything-is-a-phrase" approach which means that every term-level n-gram of an ordered query forms a phrase. Proximity terms relax phrases to term set occurrences in a text window. All these approaches share the problem that queries are not length-limited and consequently phrases and proximity terms are not length-limited either. While the *idf* values of phrases could be estimated by aggregating *idf* values stored as meta-information per term list, we do not see a way to cast the scoring model into term or term pair lists without keeping and reading posting lists with position information of individual terms which would prevent precomputation of score bounds. Position lists are required to assess whether phrase terms occur adjacently to each other or in a text window, respectively, and not only to clarify the final score value.

Like for Uematsu et al., an approximation comparable to Tonellotto et al. (cf. Section 6.2.2) could be used to compute an upper bound for tf values of a phrase p: $tf(p,d)$ cannot be greater than the minimum of $tf(t,d)$ for all single terms t in p. The accuracy of the upper bound could be improved by usage of term pair lists instead. This would alleviate the problem, but not solve it.

7.4.3 Language Models with Proximity Components

This subsection discusses the feasibility of two language models with proximity components for top-k query processing.

Lv and Zhai: Lv and Zhai's approach can be adapted to top-k query processing with early termination. Ranking documents comes in three variants which all aggregate scores at document positions. The score at position i in d is defined as

$$\begin{aligned}
S(d,q,i) &= -\sum_{t\in V} p(t|q)\cdot\log\frac{p(t|q)}{p(t|d,i)} = -\sum_{t\in V} p(t|q)\cdot\log p(t|q) + \sum_{t\in V} p(t|q)\cdot\log p(t|d,i)\\
&= -\sum_{t\in V}\frac{qtf(t)}{|q|}\cdot\log\frac{qtf(t)}{|q|} + \sum_{t\in V} p(t|q)\cdot\log p(t|d,i)\\
&\propto \sum_{t\in V} p(t|q)\cdot\log p(t|d,i) = \sum_{t\in V}\frac{qtf(t)}{|q|}\cdot\log p(t|d,i) = \sum_{t\in q}\frac{qtf(t)}{|q|}\cdot\log p(t|d,i).
\end{aligned}$$

As $-\sum_{t\in V}\frac{qtf(t)}{|q|}\cdot\log\frac{qtf(t)}{|q|}$ is constant for all documents given a query q, it can be omitted without influencing the document ranking. We further can restrict the summation to query terms, since non-query terms have no influence on $S(d,q,i)$ because their qtf value is always 0.

The positional language model (in the non-smoothed version) at position i in document d is defined as $p(t|d,i)=\frac{c'(t,i)}{\sum_{t'\in V}c'(t,i)}$, where $c'(t,i)=\sum_{j=1}^{l_d}c(t,j)\cdot k(i,j)$. Hence, $S(d,q,i)\propto\sum_{t\in q}\frac{qtf(t)}{|q|}\cdot\log\frac{c'(t,i)}{\sum_{t'\in V}c'(t,i)}$. Given a fixed kernel and spread, it is possible to store for every term t a term list with entries of the form $(d.docid, d.i, \log\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)})$, where $d.i$ is the scored position in document d, ordered by descending $\log\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)}$ values. Our solution for de Kretser and Moffat's approach described in Section 7.4.2 also assumes a fixed kernel and spread, but uses ordered term pair lists instead of term lists.

The Jelinek-Mercer smoothed variant of the positional language model is defined as

$$\begin{aligned}p_{JM}(t|d,i)&=(1-\lambda)p(t|d,i)+\lambda p(t|\mathcal{C})\\&=(1-\lambda)\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)}+\lambda\frac{ctf(t)}{l_{\mathcal{C}}}\end{aligned}$$

Hence, $S(d,q,i)\propto\sum_{t\in q}\frac{qtf(t)}{|q|}\cdot\log[(1-\lambda)\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)}+\lambda\frac{ctf(t)}{l_{\mathcal{C}}}]$.

Given a fixed kernel, spread, and weight λ, it is possible to store for every term t a term list with entries of the form $(d.docid, d.i, \log[(1-\lambda)\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)}+\lambda\frac{ctf(t)}{l_{\mathcal{C}}}])$ which is ordered by descending $\log[(1-\lambda)\frac{c'(t,i)}{\sum_{t'\in V}c'(t',i)}+\lambda\frac{ctf(t)}{l_{\mathcal{C}}}]$ values with $d.i$ being the scored position in document d. $\frac{qtf(t)}{|q|}$ is only known at query processing time and is factorized into the score by simple multiplication by the indexed scores.

The positional language model with Dirichlet prior smoothing is defined as

$$\begin{aligned}p_{DP}(t|d,i)&=\frac{c'(t,i)+\mu p(t|\mathcal{C})}{\sum_{t'\in V}c'(t',i)+\mu'}=\frac{c'(t,i)+\mu\frac{ctf(t)}{l_{\mathcal{C}}}}{\sum_{t'\in V}c'(t',i)+\mu'}=\frac{\frac{c'(t,i)\cdot l_{\mathcal{C}}+\mu\cdot ctf(t)}{l_{\mathcal{C}}}}{\sum_{t'\in V}c'(t',i)+\mu'}\\&=\frac{c'(t,i)\cdot l_{\mathcal{C}}+\mu\cdot ctf(t)}{l_{\mathcal{C}}\cdot(\sum_{t'\in V}c'(t',i)+\mu')}.\end{aligned}$$

Hence, $S(d,q,i)\propto\sum_{t\in q}\frac{qtf(t)}{|q|}\cdot\log\frac{c'(t,i)\cdot l_{\mathcal{C}}+\mu\cdot ctf(t)}{l_{\mathcal{C}}\cdot(\sum_{t'\in V}c'(t',i)+\mu')}$.

Given a fixed kernel and spread, we propose to store term lists for every term t with entries of the form $(d.docid, d.i, \log\frac{c'(t,i)\cdot l_{\mathcal{C}}+\mu\cdot ctf(t)}{l_{\mathcal{C}}\cdot(\sum_{t'\in V}c'(t',i)+\mu')})$, where $d.i$ is the scored position in document d, ordered by descending $\log\frac{c'(t,i)\cdot l_{\mathcal{C}}+\mu\cdot ctf(t)}{l_{\mathcal{C}}\cdot(\sum_{t'\in V}c'(t',i)+\mu')}$ values.

In the following, we elaborate on *bestscore* and *worstscore* bounds for documents when Dirichlet prior smoothing is used which provides the best results in the studies of Lv and Zhai. We detail score bounds for each of the three ranking options suggested by

Lv and Zhai. While $S(d,i)$ denotes the set of term lists where an entry for document d at position i has been encountered, $\bar{S}(d,i)$ represents the set of term lists which still contain unread entries and where d has not been seen yet. We define $f(d,i,t) = \log \frac{c'(t,i)\cdot l_C + \mu \cdot ctf(t)}{l_C \cdot (\sum_{t' \in V} c'(t',i) + \mu')}$.

If we score all documents by the best position in that document, the score bounds are

$$worstscore(d) = max_{i \in \{1,\dots,l_d\}}\{0, \sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j)\} \text{ and}$$

$$bestscore(d) = max_{i \in \{1,\dots,l_d\}}\{ \sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j) + \sum_{L_j \in \bar{S}(d,i)} \frac{qtf(t_j)}{|q|} high_j\}.$$

If we score all documents by the average of the best k positions in that document, we get

$$worstscore(d) = \frac{1}{k} \cdot \sum_{\substack{i \in \text{top-}k \text{ of} \\ S(d,q,\cdot) worstscores}} max\{0, \sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j)\} \text{ and}$$

$$bestscore(d) = \frac{1}{k} \cdot \sum_{\substack{i \in \text{top-}k \text{ of} \\ S(d,q,\cdot) bestscores}} \sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j) + \sum_{L_j \in \bar{S}(d,i)} \frac{qtf(t_j)}{|q|} \cdot high_j.$$

If we score all documents using a weighted score based on various spreads β_σ with $\sigma \in R$, i.e., $S(d,q) = \sum_{\sigma \in R} \beta_\sigma \cdot max_{i \in \{1,\dots,l_d\}}\{S_\sigma(d,q,i)\}$, we obtain

$$worstscore(d) = max_{i \in \{1,\dots,l_d\}}\{0, \sum_{\sigma \in R} \beta_\sigma \cdot \sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j)\} \text{ and}$$

$$bestscore(d) = max_{i \in \{1,\dots,l_d\}}\{\sum_{\sigma \in R} \beta_\sigma \cdot (\sum_{L_j \in S(d,i)} \frac{qtf(t_j)}{|q|} f(d,i,t_j) + \sum_{L_j \in \bar{S}(d,i)} \frac{qtf(t_j)}{|q|} high_j)\}.$$

To safely stop, the following inequation must be fulfilled for not yet seen (virtual) documents and holds for every score variant:

$$\sum_{L_j \in L} \frac{qtf(t_j)}{|q|} high_j < min\text{-}k,$$

where L represents the set of lists with remaining unread entries. The left part of the inequation represents the bestscore of not yet seen documents. As a virtual document has not been encountered in any dimension, for every *bestscore* computation, we can ignore the part which sums up contributions over $L_j \in S(d,i)$. Instead we expect to see virtual documents in all lists with remaining unread entries.

When scoring documents by the best position, the virtual document d has the following bestscore: $bestscore(d) = max_{i \in \{1,\dots,l_d\}}\{\sum_{L_j \in L} \frac{qtf(t_j)}{|q|} high_j\}$. If we assume that all $high_j$ point to the same (d,i) pair, we obtain the highest possible *bestscore* for d which corresponds to the left side of the inequation.

When scoring documents by the average of the best k positions in that document, the virtual document d has the following bestscore:

$$bestscore(d) = \frac{1}{k} \cdot \sum_{\substack{i \in \text{top-}k \text{ of} \\ bestscores \text{ at positions in } d}} \sum_{L_j \in \bar{S}(d,i)} \frac{qtf(t_j)}{|q|} \cdot high_j.$$

If we assume that all $high_j$ point to the same (d, i) pair (with (d, i) as top-1 bestscore) and the remaining k-1 $bestscore$ positions in d have the same bestscore as (d, i), we obtain the highest possible $bestscore$ for d. Hence, $bestscore(d) = \frac{1}{k} \cdot k \cdot \sum_{L_j \in L} \frac{qtf(t_j)}{|q|} \cdot high_j$ which corresponds to the left side of the inequation.

When scoring documents using a weighted score based on various spreads β_σ with $\sigma \in R$, we obtain:

$$bestscore(d) = max_{i \in \{1,\dots,l_d\}} \{ \sum_{\sigma \in R} \beta_\sigma \cdot (\sum_{L_j \in L} \frac{qtf(t_j)}{|q|} high_j) \}.$$

With $\sum_{\sigma \in R} \beta_\sigma = 1$ and assuming again that all $high_j$ point to the same (d, i) entry, we obtain the left side of the inequation.

Zhao and Yun: We can cast Zhao and Yun's retrieval model into index lists to allow top-k query processing with early termination. To this end, we first transform the score into indexable components:

$$\begin{aligned}
score(d,q) &= \sum_{\substack{tf(t_i,d) > 0, \\ t_i \ in \ q}} p(t_i|\hat{\theta}_q) \log \frac{p_s(t_i|d,u)}{\alpha_d \cdot p(t_i|\mathcal{C})} + \log \alpha_d \\
&= \sum_{\substack{tf(t_i,d) > 0, \\ t_i \ in \ q}} \frac{qtf(t_i)}{|q|} \log (\frac{\frac{tf(t_i,d) + \lambda Prox_B(t_i) + \mu \cdot \frac{ctf(t_i)}{l_\mathcal{C}}}{l_d + \sum_{i=1}^{|V|} \lambda Prox_B(t_i) + \mu}}{\frac{\mu}{l_d + \sum_{i=1}^{|V|} \lambda Prox_B(t_i) + \mu} \cdot \frac{ctf(t_i)}{l_\mathcal{C}}}) + \log \alpha_d \\
&= \sum_{\substack{tf(t_i,d) > 0, \\ t_i \ in \ q}} \frac{qtf(t_i)}{|q|} \log (\frac{l_\mathcal{C} \cdot (tf(t_i,d) + \lambda Prox_B(t_i) + \mu \cdot \frac{ctf(t_i)}{l_\mathcal{C}})}{\mu \cdot ctf(t_i)}) \\
&+ \log \frac{\mu}{l_d + \sum_{i=1}^{|V|} \lambda Prox_B(t_i) + \mu} \\
&\approx \sum_{\substack{tf(t_i,d) > 0, \\ t_i \ in \ q}} \frac{qtf(t_i)}{|q|} \log (\frac{tf(t_i,d) \cdot l_\mathcal{C}}{\mu \cdot ctf(t_i)} + \frac{\lambda Prox_B(t_i) \cdot l_\mathcal{C}}{\mu \cdot ctf(t_i)} + 1) \\
&+ \log \frac{\mu}{l_d + \sum_{t \in q} \lambda Prox_B(t) + \mu}.
\end{aligned}$$

$\log \frac{\mu}{l_d + \sum_{i=1}^{|V|} \lambda Prox_B(t_i) + \mu} \approx \log \frac{\mu}{l_d + \sum_{t \in q} \lambda Prox_B(t) + \mu}$ as $Prox_B(t)$ becomes very small for non-query terms in V.

For each term t_i, we maintain a term list which keeps entries of the form ($d.docid$, $\frac{tf(t_i,d)l_{\mathcal{C}}}{\mu \cdot ctf(t_i)}$), ordered by descending $\frac{tf(t_i,d)l_{\mathcal{C}}}{\mu \cdot ctf(t_i)}$ values.

For each term pair $\{t_i, t_j\}$, like for Tao and Zhai's scoring models, we keep a list with $(d.docid, mindist(t_i, t_j, d))$ entries.

Analogously to the approach we proposed for Tao and Zhai's retrieval model, term pair lists may be sorted either by descending or ascending score since we will see in the following descriptions, that we need both $high_j$ and low_j scores to compute score bounds which is different from typical scenarios involving the NRA where lists are ordered by descending score. If the list is sorted by descending score, $high_j$ is the score at the current scan position and low_j the lowest score available in a list, respectively. If the list is sorted by ascending score, $high_j$ is the highest score available in a list and low_j the score at the current scan position in a list, respectively.

For a given query, L_{cscore} is the set of term lists. $L_{pscore}(t_i)$ stands for the set of not completely read term pair lists for t_i and a different query term. $R_p(d, t_i)$ denotes the set of completely read term pair lists where d was not encountered, one partner term is t_i and the other one a different query term. While $R_p(t_i)$ denotes the set of completely read term pair lists, $\bar{R}_p(t_i)$ denotes the set of not yet completely read term pair lists, where one partner term is t_i and the other one a different query term. $S_p(d, t_i)$ represents the set of term pair lists for t_i and a different query term, where d has been encountered. $\bar{S}_p(d, t_i)$ is the set of not completely read term pair lists for t_i and a different query term, where d has not yet been encountered.

If we use minimum distance as proximate centrality, i.e., $Prox_{MinDist}$, we obtain the following score bounds for the proximity component in the non-α_d part:

$$worstscore_p(t_i, d) = \frac{\lambda x^{-min(l_d, min_{L_{ij} \in S_p(d,t_i)}(mindist(t_i,t_j,d)))} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)} \text{ and}$$

$$bestscore_p(t_i, d) = \frac{\lambda x^{-min(l_d, min_{L_{ij} \in S_p(d,t_i)}(mindist(t_i,t_j,d)), min_{L_{ij} \in \bar{S}_p(d,t_i)}(low_{ij}))} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)}.$$

$worstscore_p(t_i, d)$ gets smallest if the exponent of x is as negative as possible. This is the case if d is not contained in any list $L_{ij} \in \bar{S}_p(d, t_i)$ which would result in $Dis(t_i, t_j, d) = l_d$. For those lists where d has been encountered we have to set $mindist(t_i, t_j, d)$ as distance. $bestscore_p(t_i, d)$ gets largest if the exponent of x is as little negative as possible. To this end, we expect to see d in L_{ij} from $\bar{S}_p(d, t_i)$ having the smallest low_{ij} value. For $worstscore_{p,\alpha_d}(q)$ and $bestscore_{p,\alpha_d}(q)$ the actions to take to minimize/maximize the score, respectively, are switched since x does not stand in the numerator, but in the denominator.

Consequently, for α_d, we obtain the following score bounds:

$$worstscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-min(l_d, min_{L_{ij} \in S_p(d,t_i)}(mindist(t_i,t_j,d)), min_{L_{ij} \in \bar{S}_p(d,t_i)}(low_{ij}))} + \mu}$$

$$\text{and } bestscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-min(l_d, min_{L_{ij} \in S_p(d,t_i)}(mindist(t_i,t_j,d)))} + \mu}.$$

To safely stop, the following inequation must be fulfilled for not yet seen (virtual) documents:

$$\sum_{L_i \in L_{cscore}} \frac{qtf(t_i)}{|q|} \cdot \log(high_i + \frac{\lambda x^{-min(l_d, min_{L_{ij} \in L_{pscore(t_i)}}(low_{ij}))} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)} + 1)$$

$$+ \log \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-l_d}(t_i) + \mu} < min\text{-}k.$$

$high_i$ is the highest score in the term list of t_i which becomes 0 if the list has been read completely. As the left side of the inequality is to be maximized, we expect for the non-α_d-part that we see d with the lowest possible low_{ij} score. If all term pair lists have been read, we have to use l_d. For the α_d-part, we expect d not to be seen in any of the remaining term pair lists and hence employ l_d.

If we use average distance as proximate centrality, i.e., $Prox_{AvgDist}$, we get the following score bounds for the proximity component in the non-α_d part:

$$worstscore_p(t_i, d) = \frac{\lambda x^{-max_{A \subseteq \bar{S}_p(d,t_i)}[f(S_p(d,t_i),A)(g(S_p(d,t_i)) + \sum_{L_{ij} \in A} high_{ij} + \sum_{L_{ij} \in R_p(d,t_i)} l_d)]} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)}$$

and

$$bestscore_p(t_i, d) = \frac{\lambda x^{-min_{A \subseteq \bar{S}_p(d,t_i)}[f(S_p(d,t_i),A)(g(S_p(d,t_i)) + \sum_{L_{ij} \in A} low_{ij} + \sum_{L_{ij} \in R_p(d,t_i)} l_d)]} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)},$$

where $f(S_p(d, t_i), A) = \frac{1}{|S_p(d,t_i)|+|A|-1}$ and $g(S_p(d, t_i)) = \sum_{L_{ij} \in S_p(d,t_i)} mindist(t_i, t_j, d)$.

For the computation of $worstscore_p(t_i, d)$, we again aim at making the exponent of x as negative as possible. To this end, we negatively maximize it over all subsets A of $\bar{S}_p(d, t_i)$; we expect d to be encountered in all lists in A with the highest possible value $high_{ij}$. Hence, $f(S_p(d, t_i), A) = \frac{1}{n-1}$ where $n = |S_p(d, t_i)| + |A|$. For $bestscore_p(t_i, d)$, we make the exponent of x as little negative as possible. Therefore, we expect d to be encountered in all lists in A with the lowest possible value low_{ij}.

For α_d, we obtain the following score bounds:

$$worstscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-min_{A \subseteq \bar{S}_p(d,t_i)}[f(S_p(d,t_i),A)(g(S_p(d,t_i)) + \sum_{L_{ij} \in A} low_{ij} + \sum_{L_{ij} \in R_p(d,t_i)} l_d)]} + \mu}$$

and

$$bestscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-max_{A \subseteq \bar{S}_p(d,t_i)}[f(S_p(d,t_i),A)(g(S_p(d,t_i)) + \sum_{L_{ij} \in A} high_{ij} + \sum_{L_{ij} \in R_p(d,t_i)} l_d)]} + \mu}.$$

$worstscore_{p,\alpha_d}(q)$ and $bestscore_{p,\alpha_d}(q)$ are handled analogously to the other proximate centrality variants.

To safely stop, the following inequation must hold for not yet seen (virtual) documents:

$$\sum_{L_i \in L_{cscore}} \frac{qtf(t_i)}{|q|} \cdot \log(high_i + \frac{\lambda x^{-min_{A \subseteq \bar{R}_p(t_i)}[\frac{1}{|A|-1}(\sum_{L_{ij} \in A} low_{ij} + \sum_{L_{ij} \in R_p(t_i)} (min_{d \in \mathcal{C})} l_d)]} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)} + 1)$$

$$+ \log \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-max_{A \subseteq \bar{R}_p(t_i)}[\frac{1}{|A|-1} \cdot (\sum_{L_{ij} \in A} high_{ij} + \sum_{L_{ij} \in R_p(t_i)} (max_{d \in \mathcal{C}} l_d))]} + \mu} < min\text{-}k.$$

$high_i$ is the highest score in the term list of t_i which becomes 0 if the list has been read completely. For the non-α_d−part, we aim at rendering the exponent of x as little negative as possible: to this end, we use low_{ij} for all lists in A and the minimum document length in the collection for completely read lists. For the α_d−part, we aim at rendering the exponent as negative as possible. Therefore, we use $high_{ij}$ for lists in A and the maximum length of any document in the collection for completely read lists.

If we use summed distance as proximate centrality, i.e., $Prox_{SumDist}$, we can work with the following score bounds for the proximity component in the non-α_d part:

$$worstscore_p(t_i, d) = \frac{\lambda x^{-(g(S_p(d,t_i)) + \sum_{L_{ij} \in R_p(d,t_i)} l_d + \sum_{L_{ij} \in \bar{S}_p(d,t_i)} l_d)} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)} \text{ and}$$

$$bestscore_p(t_i, d) = \frac{\lambda x^{-(g(S_p(d,t_i)) + \sum_{L_{ij} \in R_p(d,t_i)} l_d + \sum_{L_{ij} \in \bar{S}_p(d,t_i)} low_{ij})} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)},$$

where $g(S_p(d, t_i)) = \sum_{L_{ij} \in S_p(d,t_i)} mindist(t_i, t_j, d)$.

$worstscore_p(t_i, d)$ gets smallest if the exponent of x is as negative as possible. To accomplish that we set l_d for all lists where d has not been encountered (i.e., lists in $R_p(d, t_i)$ or $\bar{S}_p(d, t_i)$) which means that d will not be seen in those lists any more. $bestscore_p(t_i, d)$ gets largest if the exponent of x is as little negative as possible. For each list L_{ij} in $\bar{S}_p(d, t_i)$ we set low_{ij} to fulfill this goal.

$worstscore_{p,\alpha_d}(q)$ and $bestscore_{p,\alpha_d}(q)$ are handled analogously to the other proximate centrality variants.

For α_d, we obtain the following score bounds:

$$worstscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-(g(S_p(d,t_i)) + \sum_{L_{ij} \in R_p(d,t_i)} l_d + \sum_{L_{ij} \in \bar{S}_p(d,t_i)} low_{ij})} + \mu} \text{ and}$$

$$bestscore_{p,\alpha_d}(q) = \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-(g(S_p(d,t_i)) + \sum_{L_{ij} \in R_p(d,t_i)} l_d + \sum_{L_{ij} \in \bar{S}_p(d,t_i)} l_d)} + \mu}.$$

To safely stop, the following inequation must be fulfilled for not yet seen (virtual) documents:

$$\sum_{L_i \in L_{cscore}} \frac{qtf(t_i)}{|q|} \cdot \log(high_i + \frac{\lambda x^{-(\sum_{L_{ij} \in L_{pscore}(t_i)} (low_{ij}) + \sum_{L_{ij} \in R_p(t_i)} (min_{d \in \mathcal{C}} l_d))} \cdot l_{\mathcal{C}}}{\mu \cdot ctf(t_i)} + 1)$$

$$+ \log \frac{\mu}{l_d + \sum_{t_i \in q} \lambda x^{-(\sum_{L_{ij} \in R_p(t_i)} (max_{d \in \mathcal{C}} l_d) + \sum_{L_{ij} \in L_{pscore}(t_i)} (max_{d \in \mathcal{C}} l_d))} + \mu} < min\text{-}k.$$

$high_i$ is the highest score in the term list of t_i which becomes 0 if the list has been read completely. For the non-α_d-part, (to render the exponent of x as little negative as possible) we expect for the not yet completely read term pair lists that the virtual document is encountered with the lowest possible value low_{ij}. For the completely read term pair lists in $R_p(t_i)$ we use the minimum length of any document in the collection. For the α_d-part, we aim at rendering the exponent as negative as possible. Therefore, we use the maximum length of any document in the collection.

7.4.4 Learning to Rank

In this subsection, we discuss the feasibility of some Learning to Rank approaches for early termination in an NRA setting.

Svore et al.: Svore et al.'s approach is based on Song et al.'s work which incorporates assessing the goodness of espans. If the learned scoring model involves this kind of features, it shares the same problems (cf. Section 7.4.2). This also applies to proximity match features which require knowledge about spans in each document. Non-span related features, however, can be cast into term lists (λBM25 features) or term pair lists (λBM25-2 features) and make early termination possible.

Metzler and Croft: In principle, Metzler and Croft's retrieval model can be cast into score-ordered index lists to make early termination possible. For each of the three kinds of potential functions, we need a separate list with docids and their feature values ordered by descending feature value:

- For single terms q_i, a term list for q_i could store entries of the form

$$(D.docid, \log[(1-\alpha_D)\frac{tf(q_i, D)}{l_D} + \alpha_D\frac{ctf(q_i)}{l_\mathcal{C}}]).$$

- For ordered potential functions representing phrases "$q_i, \ldots, q_{i+k}$", we could store entries of the form $(D.docid, \log[(1-\alpha_D)\frac{tf_{\#1(q_i,\ldots,q_{i+k}),D}}{l_D} + \alpha_D\frac{ctf_{\#1(q_i,\ldots,q_{i+k})}}{l_\mathcal{C}}])$.

- For unordered potential functions representing ordered or unordered occurrences of query term sets $\{q_i, \ldots, q_j\}$, we could store entries of the form

$$(D.docid, \log[(1-\alpha_D)\frac{tf_{\#uwN(q_i,\ldots,q_j),D}}{l_D} + \alpha_D\frac{ctf_{\#uwN(q_i,\ldots,q_j),D}}{l_\mathcal{C}}]).$$

The sum of the weighted scores from the three potential functions can be used to compute score bounds during query processing. As usual, storing entries for single terms is not an issue. The problem that prevents this approach from being practical for top-k query processing, without restrictions on the queries, is the huge amount of possible phrases and sets of query terms that may occur within a text window.

Cummins and O' Riordan: Cummins and O' Riordan use genetic programming to learn a scoring model.

The baseline scores can be cast into term lists. $score_{ES}$ is materialized analogously to $score_{\mathrm{BM25}}$ as described for Rasolofo and Savoy's approach. A term list for term t_i could keep an entry of the form $(d.docid, score_{\mathrm{ES}}(d, t_i)/qtf(t_i))$ for each document d with at least one occurrence of term t_i where the list is ordered by descending $score_{\mathrm{ES}}(d, t_i)/qtf(t_i)$ values.

The learned retrieval model is feasible for top-k query processing if only measures are used which can be cast into term pair lists. These measures include measures 1 to 8 listed in Section 2.7.4. The document length (measure 12) is known at indexing time so that it can also be incorporated into precomputed scores. The number of unique query terms in a document (measure 13) can be captured by both reading term and term pair lists. For learned scoring models where we can factor out the qt-part of the original score, we could follow the approach we have proposed for making the *AvgDist* measure by Tao and Zhai feasible for top-k querying. While scanning the lists, a document's qt value ranges from the number of distinct query terms seen so far for that document to the number of query terms: while the *bestscore* would maximize the score value, the $worstscore$ minimizes the score value over all possible qt values. Having completely read a $score_{\mathrm{ES}}$ or $score_{\mathrm{BM25}}$ based term list can shrink the range of qt.

Measures 10 and 11 (i.e., $FullCover$ and $MinCover$) correspond to measures used by Tao and Zhai ($FullCover$ corresponds to *Span* in Tao and Zhai's paper) and are inherently query-dependent and cannot be easily decomposed into term pair lists. More details can be found in the paragraph about Tao and Zhai's approach in Section 7.4.1.

7.4.5 Summary

We have shown that, for a surprisingly high fraction of proximity score-enhanced retrieval models, it is possible to cast them into precomputed term and term pair lists:

Rasolofo and Savoy's scoring model can be cast into precomputed term and term pair lists similar to our modification of Büttcher's approach. Tao and Zhai's approach is feasible for index precomputation when one of the three distance aggregation measures is used. However, different from a conventional NRA strategy that orders lists by decreasing impact, here, term pair lists may be ordered by ascending impact since both low and high values of lists are needed to determine worstscores and bestscores of documents. Thus, it may be worthwhile thinking about reading from both ends of the lists for faster termination. De Kretser and Moffat's second algorithm can be cast into term pair lists with document-position related scores although the index may take up too much space to be practical; the maximum score over all positions becomes the document score. Lv and Zhai's approach can be cast into term lists that keep scores for (document, position) pairs. Zhao and Yun's approach keeps three kinds of index lists: term lists for the content score part, term pair lists for the proximity component and one query-independent document-constant list. Cummins and O'Riordan's approach can be used for top-k query processing if we restrict learned scores to consist only of components that can be cast into term pair lists.

Some scoring models cannot be cast into index lists due to space reasons: Tao and Zhai's model with span-based measures (*Span* and $MinCover$) requires position information at query processing time. Precomputing index lists for Monz' approach is problematic since minimum spans are inherently query-load dependent. The approaches by Song et al. and Svore et al. rely on espans which can be determined only if we know, for any query term occurrence in a document, which query term follows next in the document and at which position. The approaches suggested by Mishne and de Rijke and by Metzler and Croft rely on tf values of phrasal occurrences and term set occurrences within text windows, respectively. Precomputing and storing them for arbitrary n-grams exceeds reasonable space requirements.

Chapter 8

Index Tuning for High-Performance Query Processing

The first part of this chapter introduces a joint framework for trading off index size and result quality. It provides optimization techniques for tuning precomputed indexes towards either maximal result quality or maximal query processing performance under controlled result quality, given an upper bound for the index size. The framework allows to selectively materialize lists for pairs based on a query log to further reduce the index size. Extensive experiments with two large text collections demonstrate runtime improvements of more than one order of magnitude over existing text-based processing techniques with reasonable index sizes. This part is based on our article published in [BS12] and enriched with results from our participation in the INEX 2009 Ad Hoc and Efficiency Tracks [BS09] and the TREC 2010 Web Track [BS10].

The second part of this chapter introduces a new index structure to improve cold cache performance by reducing the number of fetched lists traded in for more read bytes. This part is based on our work presented in [BS11].

8.1 Introduction

8.1.1 Motivation

In Chapter 7 we have presented an approach to integrate proximity scores as an integral part of query processing. This showed that proximity scores can improve not only result quality, but also efficiency, by means of index pruning. However, the index parameters for pruning were chosen in an ad hoc manner, lacking systematic optimization. We now extend results from Chapter 7 towards a configurable indexing framework which can be tuned either for maximal and dependable query performance under result quality control or for maximal result quality given an index size budget. Existing methods for the integration of proximity scores into efficient query processing algorithms for quickly computing the best k results (e.g., [CCKS07, PRL$^+$07]) make use of precomputed lists of documents where tuples of terms, usually pairs, occur together, usually incurring a huge index size compared to term-only indexes, or focusing on conjunctive queries only. There are existing techniques for lossy index compression that materialize only a subset of all term pairs, e.g., those term pairs occurring in queries of a query log. In contrast and orthogonally to these techniques, this chapter aims at limiting the size of each term pair list by limiting the maximal list length and imposing a minimal proximity score per tuple in a term pair list. At the same time, the choice of term pair index lists to be materialized can be based on frequent queries in a query log. Our method can be tuned towards either guaranteeing maximal result quality or maximal query performance at controlled result quality within a given index size constraint. For both optimization goals, the result of the method is a set of pruned index lists of a fixed maximal length, which means that the worst-case cost for evaluating a query with this index can be tightly bound as well. In our experiments with the GOV2 collection (reported in Section 8.5), we show that 310 entries per list can be enough to give the same result quality as a standard score taking only term frequencies into account. We have measured an average warm cache retrieval time of less than 30ms at a cache size of just 64MB for a standard query load of 50,000 queries, an average cold cache retrieval time of 127ms and a hot cache retrieval time of less than 1ms. In this configuration, the size of the compressed index is 95GB, only slightly larger than the compressed collection. Similar query processing costs can be achieved for much larger collections, such as the recent ClueWeb09 collection.

8.1.2 Contributions

This chapter makes the following important contributions:

- It introduces a tunable indexing framework for terms and term pairs for optimizing index parameters towards either maximal result quality or maximal query processing performance under result quality control, given a maximal index size.
- It allows a selective materialization of term pair index lists based on information from a query log.
- The resulting indexes provide dependable query execution times while providing a result quality comparable to or even better than unpruned term indexes.
- It experimentally demonstrates that the resulting index configurations allow query processing that yields almost one order of magnitude performance gain compared to a state-of-the-art top-k algorithm while returning results of at least comparable quality.

8.1.3 Outline of the Chapter

The remainder of this chapter is structured as follows. Section 8.2 elaborates on the index organization and the employed index compression techniques. Section 8.3 presents the index tuning framework within the MapReduce paradigm and formulates tuning as an optimization problem that considers both index size and retrieval quality. Section 8.4 shows how the size of the index can be reduced further using a query log. Section 8.5 experimentally evaluates our index tuning techniques from Section 8.3 with two large text collections from TREC, namely GOV2 and ClueWeb09 (cf. Section 3.2.1), for different result size cardinalities. It can be tuned either towards effectiveness or efficiency, given a size limit for the pruned indexes, both in the presence and absence of relevance assessments. We compare the query processing performance of merge joins with pruned indexes as input to a state-of-the-art document-at-a-time algorithm that uses dynamic pruning on unpruned indexes and provide additional results for a proximity-enhanced variant of that state-of-the-art document-at-a-time algorithm. Query processing performance is measured both by abstract measures and average query processing times for different cache settings. Furthermore, we evaluate the effect of query log-based combined list pruning. Additional results with ClueWeb09 demonstrate the scalability of our index tuning approach. As a third collection we use the textual content of the INEX Wikipedia collection from 2009 (cf. Section 3.2.2). We present results from our participation in the INEX 2009 Efficiency and Ad Hoc Tracks. Section 8.6 presents a novel hybrid index structure that accelerates cold cache query processing, trading off a reduced number of index lists for an increased number of bytes to read.

Please note that the techniques described in this chapter are not limited to this particular proximity scoring model (cf. Section 7.2.2) we use throughout the chapter: whenever bigram features, representing a proximity score contribution, can be stored in term pair index lists, our techniques can be applied as well. In Section 7.4 which investigates the feasibility of various proximity scoring models for top-k query processing, we have described which approaches can be cast into term pair index lists.

8.2 Indexes

Our studies in Section 7.3 have shown that by means of pruned term and combined index lists as input to a `TL+CL` processing strategy, we can achieve the best retrieval quality among the presented processing approaches for many pruning levels. We have described the abstract layout of these two index structures in Section 7.3.1 and will now discuss the efficient physical implementation of these index structures.

The index tuning framework described in this chapter transparently supports all kinds of index compression. We will now introduce our proof-of-concept implementation of index compression which applies delta and v-byte encoding [CMS10, ZM06]; we did not perform any specific optimization for the parameters, for example the number of bits to represent a score, but we think that the values we chose are reasonable.

Our inverted lists are usually sorted by docid, but may also be sorted by descending score ($score_{\mathrm{BM25}}$ for term lists, acc_d for combined lists).

Due to the implementation of our tuning framework (cf. Section 8.3.2) which parallelizes the indexing process across a cluster of servers, each index list is assigned to one of several partitions. Figure 8.1 depicts the general structure of our term list index.

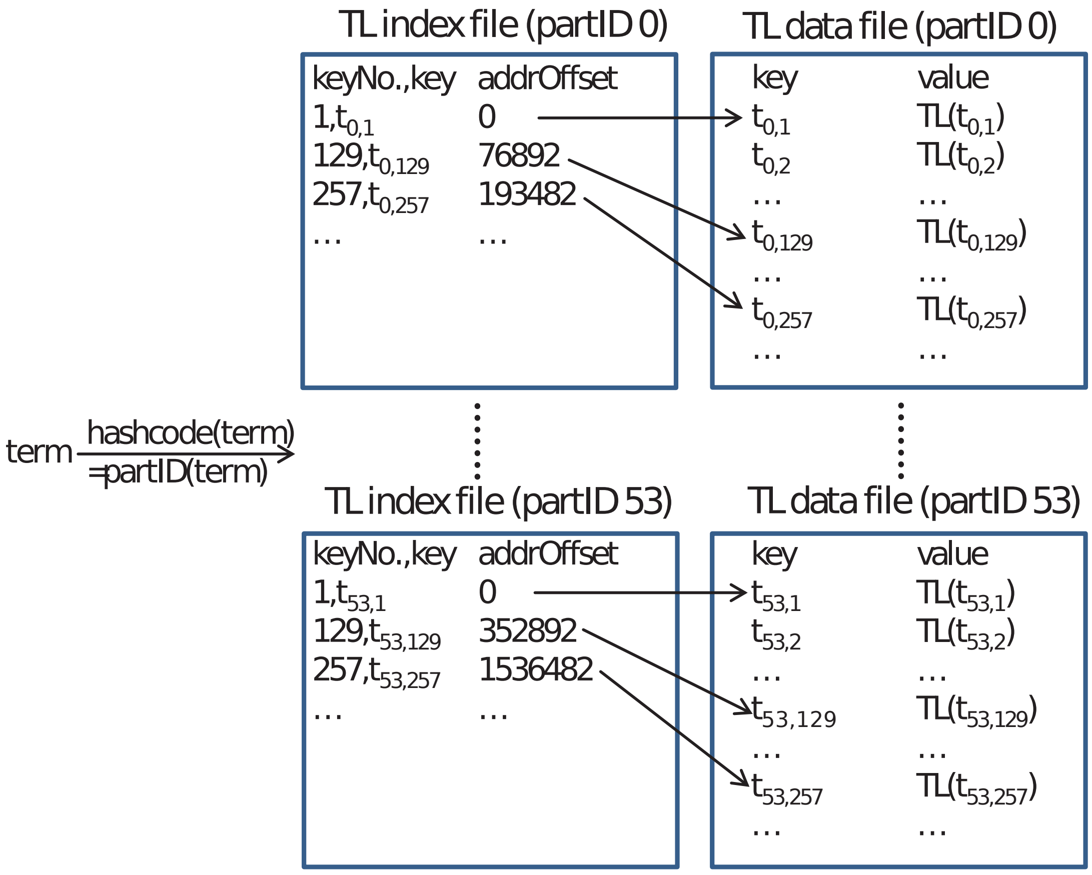
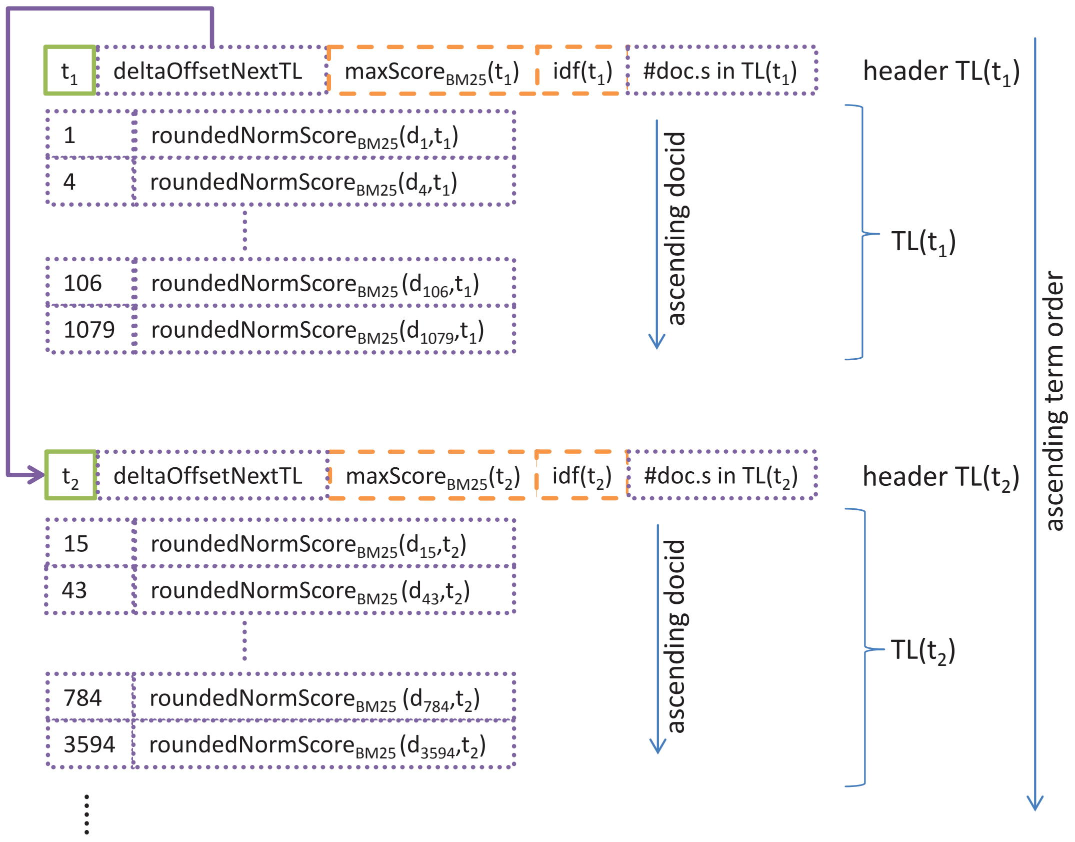


Figure 8.1: Index and data files for TLs.

The hashcode of a term determines the partition where its term list (TL) is stored. For each partition we generate one index file and one data file. All data files together contain the complete index information and consist of key, value pairs: each key is a term whose value is its TL. The index files are used to find the start address to lookup in the data file where a term's TL may be stored. Each index file stores every k^{th} key, where the keys are stored in ascending lexicographical order. Every key in an index file is assigned to an address offset (in bytes) which points to the position in the data file of the corresponding partition where the key and its TL are stored. The access structure to find the inverted list for a given key is implemented analogously to that of MapFiles in Hadoop [Whi09]; again, this is just a proof-of-concept implementation, we could alternatively have implemented the access structure with B+-trees, for example.

In the system's initialization phase, before processing any query, all index files are loaded into main memory. To locate the inverted list for a key, we first determine its partition id by means of its hashcode. The key or its closest smaller neighbor key (in lexicographic order) is determined in the in-memory index using binary search, then the data file is searched linearly from the offset of that key until either the right list is found or a larger key is encountered; in the latter case, there is no list for that key in the data file, i.e., the key is not in the index.

Indexing combined lists (CLs) works analogously to indexing TLs, with the only difference that the keys are term pairs instead of terms and the values are CLs instead of TLs. Our indexes are materialized with 54 partitions and k=128 as step width, however these numbers are configurable.



t_1
deltaOffsetNextTL
$\text{maxScore}_{\text{BM25}}(t_1)$
$\text{idf}(t_1)$
#doc.s in $\text{TL}(t_1)$
header $\text{TL}(t_1)$
1
$\text{roundedNormScore}_{\text{BM25}}(d_1,t_1)$
4
$\text{roundedNormScore}_{\text{BM25}}(d_4,t_1)$
106
$\text{roundedNormScore}_{\text{BM25}}(d_{106},t_1)$
1079
$\text{roundedNormScore}_{\text{BM25}}(d_{1079},t_1)$
ascending docid
$\text{TL}(t_1)$
t_2
deltaOffsetNextTL
$\text{maxScore}_{\text{BM25}}(t_2)$
$\text{idf}(t_2)$
#doc.s in $\text{TL}(t_2)$
header $\text{TL}(t_2)$
15
$\text{roundedNormScore}_{\text{BM25}}(d_{15},t_2)$
43
$\text{roundedNormScore}_{\text{BM25}}(d_{43},t_2)$
784
$\text{roundedNormScore}_{\text{BM25}}(d_{784},t_2)$
3594
$\text{roundedNormScore}_{\text{BM25}}(d_{3594},t_2)$
ascending docid
$\text{TL}(t_2)$
ascending term order

Figure 8.2: Compressed TLs in docid-order.

We will now describe in detail how the data files are organized. Figure 8.2 and Figure 8.3 show the structure of compressed TLs and CLs stored in docid order, respectively. In both figures, we mark the encoding and data types by different kinds of lines: green solid lines indicate UTF-8 encoding (consuming two bytes plus the number of UTF-8 bytes), violet dotted lines a v-byte encoding (of flexible size), and orange dashed lines float-typed data (consuming 4 bytes each).

Figure 8.2 shows that each TL(t) is preceded by a header that contains the UTF-8 encoded term t, the v-byte encoded byte offset value to the beginning of the next TL (needed to search the right list in the data file), and the maximum $score_{\text{BM25}}$ value of TL(t) which is required to reconstruct the stored BM25 scores in the corresponding TL. Furthermore, we maintain the $idf(t)$ value that is required to process CLs as described later. Additionally, we store the number of documents for each TL. The actual TL contains a list of pairs that contain the docid and its rounded normalized BM25 score,

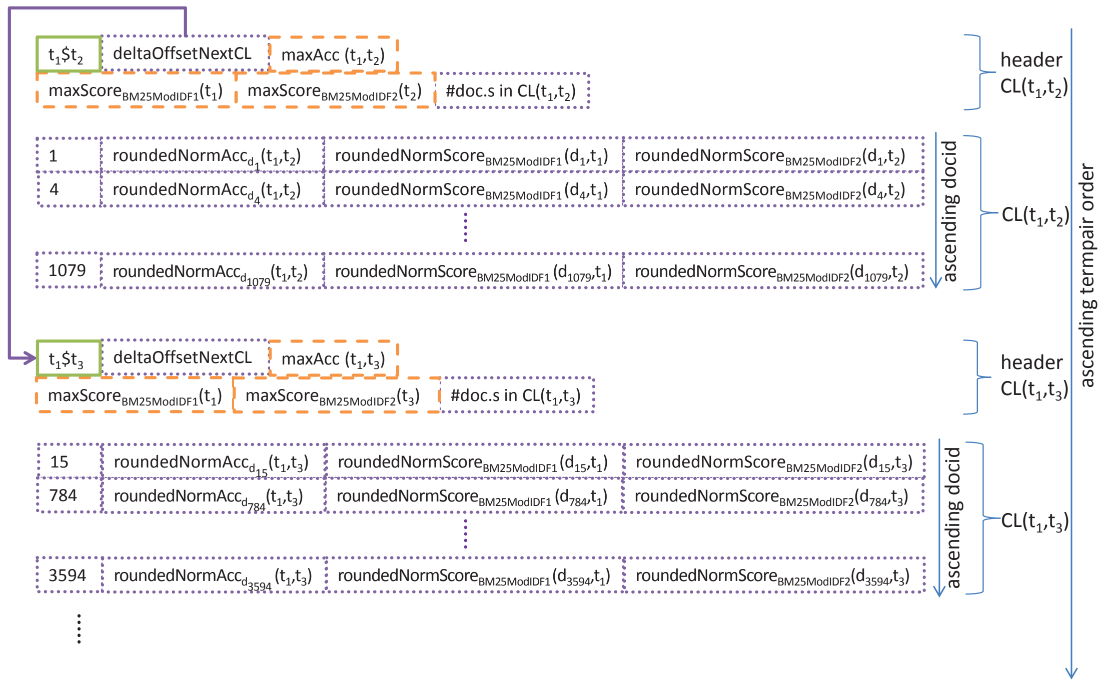

t₁$t₂
deltaOffsetNextCL
maxAcc (t₁,t₂)
maxScoreBM25ModIDF1(t₁)
maxScoreBM25ModIDF2(t₂)
#doc.s in CL(t₁,t₂)
header CL(t₁,t₂)
1
roundedNormAccd₁(t₁,t₂)
roundedNormScoreBM25ModIDF1(d₁,t₁)
roundedNormScoreBM25ModIDF2(d₁,t₂)
4
roundedNormAccd₄(t₁,t₂)
roundedNormScoreBM25ModIDF1(d₄,t₁)
roundedNormScoreBM25ModIDF2(d₄,t₂)
1079
roundedNormAccd₁₀₇₉(t₁,t₂)
roundedNormScoreBM25ModIDF1(d₁₀₇₉,t₁)
roundedNormScoreBM25ModIDF2(d₁₀₇₉,t₂)
ascending docid
CL(t₁,t₂)
t₁$t₃
deltaOffsetNextCL
maxAcc (t₁,t₃)
maxScoreBM25ModIDF1(t₁)
maxScoreBM25ModIDF2(t₃)
#doc.s in CL(t₁,t₃)
header CL(t₁,t₃)
15
roundedNormAccd₁₅(t₁,t₃)
roundedNormScoreBM25ModIDF1(d₁₅,t₁)
roundedNormScoreBM25ModIDF2(d₁₅,t₃)
784
roundedNormAccd₇₈₄(t₁,t₃)
roundedNormScoreBM25ModIDF1(d₇₈₄,t₁)
roundedNormScoreBM25ModIDF2(d₇₈₄,t₃)
3594
roundedNormAccd₃₅₉₄(t₁,t₃)
roundedNormScoreBM25ModIDF1(d₃₅₉₄,t₁)
roundedNormScoreBM25ModIDF2(d₃₅₉₄,t₃)
ascending docid
CL(t₁,t₃)
ascending termpair order

Figure 8.3: Compressed CLs in docid-order.

where $\text{roundedNormScore}_{\text{BM25}}(d,t)$ is defined as

$$round((2^{14}-1)\cdot score_{\text{BM25}}(d,t)/max_{d'\in\mathcal{C}}(score_{\text{BM25}}(d',t))).$$

As this value is in $[0,2^{14}-1]$, it can be encoded into at most 2 bytes with v-byte encoding.

Figure 8.3 shows that the header for each $\text{CL}(t_i,t_j)$ contains the UTF-8 encoded term pair string $t_i\$t_j$ ($ is the term delimiter), and the v-byte encoded byte offset value to the term pair of the next CL in the same data file. Furthermore, the header contains the maximum $acc_d(t_i,t_j)$ score in that CL named $\text{maxAcc}(t_i,t_j)$, and the maximum $score_{\text{BM25}}$ modulo *idf* values for both $score_{\text{BM25}}$ dimensions in $\text{CL}(t_i,t_j)$ named $\text{maxScore}_{\text{BM25ModIDF1}}(t_i)$ and $\text{maxScore}_{\text{BM25ModIDF2}}(t_j)$, respectively. We do not include *idf* scores in the index as they are not yet known at CL indexing time. As TLs and CLs are used in combination for query processing, the *idf* scores can be obtained from the TLs at query processing time. Like for each TL, for convenience reasons during query processing, we store the number of documents included in each CL. The actual CL contains a list of tuples; each tuple contains the docid plus three scores:

- $\text{roundedNormAcc}_d(t_i,t_j) = round((2^{14}-1)\cdot\frac{acc_d(t_i,t_j)}{max_{d'\in\mathcal{C}}(acc_{d'}(t_i,t_j))})$,
- $\text{roundedNormScore}_{\text{BM25ModIDF1}}(d,t_i) = round((2^{14}-1)\cdot\frac{\frac{score_{\text{BM25}}(d,t_i)}{idf(t_i)}}{max_{d'\in\mathcal{C}}\frac{score_{\text{BM25}}(d',t_i)}{idf(t_i)}})$,

  and

- roundedNormScore$_{\text{BM25ModIDF2}}(d, t_j) = round((2^{14}-1) \cdot \frac{\frac{score_{\text{BM25}}(d,t_j)}{idf(t_j)}}{max_{d' \in \mathcal{C}} \frac{score_{\text{BM25}}(d',t_j)}{idf(t_j)}})$.

Like in TLs, each v-byte encoded rounded normalized score does not require more than two bytes.

When both CLs and TLs are docid-ordered, the docid values in each list are first delta-encoded and then stored as v-bytes, and the score(s) of the entries are encoded as v-bytes with at most 2 bytes per score. For score order, TLs are sorted by descending $score_{\text{BM25}}$ which are delta-encoded and then stored as v-bytes, the corresponding docids are encoded as v-bytes. CLs are sorted by descending acc_d scores which are delta-encoded and then stored as v-bytes; the corresponding docids and the $score_{\text{BM25}}$ contributions for the two terms represented by the combined list are encoded as v-bytes. In score order, ties are broken using docid. While ties are rare for term lists, they are more frequent for combined lists which is due to the fact that acc_d scores as sorting criterion in combined lists are more similar than BM25 scores as sorting criterion in term lists.

8.3 Parameter Tuning

8.3.1 Tuning as Optimization Problem

We have demonstrated in Section 7.3 that using term and combined index lists together for query processing can reduce processing cost by an order of magnitude compared to using only term index lists and a standard top-k algorithm. At the same time, the proximity component of the score helps to additionally improve result quality. However, these great properties come at a big price: an index that maintains complete information for all combined lists will be several orders of magnitude larger than the original collection of documents and is therefore infeasible even for medium-sized collections. We proposed to keep only prefixes of fixed length of each list, and demonstrated that this improved both result quality and query performance while greatly reducing index size. Section 7.3 also included experiments indicating that term pair occurrences that are more than approximately 10 positions apart (runs marked with $acc_d \geq 0.01$) hardly play a role for result quality and can therefore usually be ignored. We take over this finding, so whenever we talk about term pair occurrences, we mean occurrences of different terms within a window of at most 10 positions in the same document. Note, however, that all our methods are still valid when this constraint is relaxed.

However, in Section 7.3, we did not provide any means for selecting the list length cutoff, which usually depends on the document collection and on the required result quality. There is a tradeoff between index size and quality: longer lists usually mean better results, but also a bigger index, while setting the length cutoff very low will greatly reduce index size, but at the same time also hurt result quality.

This section introduces an automated method to tune index parameters such that both the size of the resulting index and the quality of results generated using this index meet predefined requirements. (Note that for the moment, our approach keeps all combined lists, but limits the information stored in each list. We will discuss in Section 8.4 how a subset of all combined lists can be selected based on the occurrence of the pairs in a query log.) We will proceed as follows: we first define two parameters for tuning the index size, then we show how to estimate the size of an index given the tuning parameters. Next, we define measures for the quality of a pruned index, and finally, we formally define index tuning as an optimization problem and show how to solve it.

Parameters

We start with defining two parameters to tune the selection of index entries stored in each term or combined index list:

- *Minimal score cutoff*: we keep only index entries with a score that is not below a certain lower limit m.
- *List length cutoff*: we keep at most the l entries from each list that have the highest scores.

These two parameters allow us to systematically reduce the size of the resulting index with a controllable influence on result quality. Figure 8.4 shows how the index size for GOV2, relative to an unpruned index, changes with varying l and m.

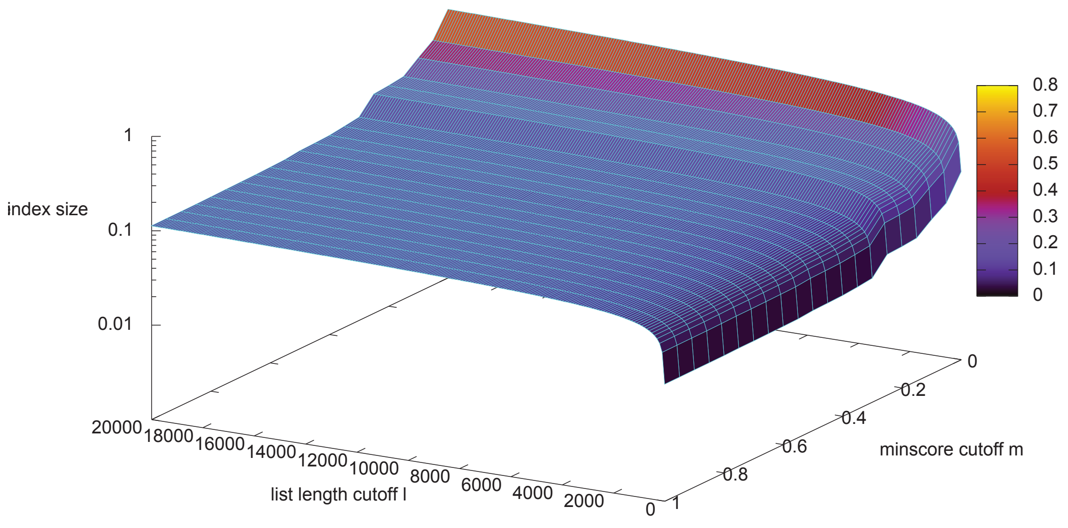


Figure 8.4: Relative index size with varying list length and minscore cutoffs.

We denote the index consisting of all term index lists for collection C by $T(C)$, and the index consisting of all term and combined index lists for C by $I(C)$. We will use the term *inverted lists* synonymously for index lists. We write $I(C, l, m)$ for the index for document collection C that consists of term and combined index lists, where each list is limited to the l entries with highest score and the combined lists contain only entries with an acc_d-score of at least m. We use the similar notation $T(C, l)$ for an index consisting of only term lists where each list contains only the l entries with highest score. Note that we do not perform score-based pruning on term lists. We omit C when the collection is clear from the context.

Index Size

An important constraint in our optimization process is the maximal storage space that the final pruned index is allowed to occupy. We will denote the *size* of an index I in bytes by $|I|$. The size of an uncompressed index depends on (1) the aggregated number $N(I)$ of index entries in all lists, (2) the size s of each index entry in bytes, (3) the number of different keys $K(I)$ (i.e., terms and/or term pairs) in the index, and (4) the per-key overhead a of the access structure to associate a key with an offset in the inverted file. For a compressed index, s is not constant, but depends on the entry and the previous entry (due to delta encoding). We can formally define the size of the index I as

$$|I| := s \cdot N(I) + a \cdot K(I).$$

This simple definition is only valid when all index lists are of the same type. In our application, we may have two different index lists, term lists and combined lists, which may differ in number of entries, number of keys, and entry size. We therefore write $N_t(I)$ for the number of term list entries in index I and $N_c(I)$ for the number of combined list entries in I, with $N(I) = N_t(I) + N_c(I)$, and use a similar notation for s and $K(I)$. The more accurate size of an index I is then

$$|I| := s_t \cdot N_t(I) + s_c \cdot N_c(I) + a \cdot (K_t(I) + K_c(I)).$$

For an uncompressed index, assuming that integers and floats need 4 bytes to store, we can set $s_t := 4 + 4 = 8$ (document ID and content score) and $s_c := 4 + 4 + 4 + 4 = 16$ (document ID, proximity score, and content scores for both terms). We can estimate a similarly (for example, by assuming that a corresponds to the average key length plus the space for a pointer into the inverted file).

We are typically interested in estimating the size of a pruned index $I(l, m)$ or $T(l)$ without actually materializing it (because materializing it takes a lot of time and the index may be too large to be completely materialized anyway). In the following we discuss how to estimate $|I(l, m)|$, the adaptation to $|T(l)|$ is straightforward. We consider only a sample P of all possible keys (i.e., terms and term pairs) and use it to approximate the distribution of list lengths, given a list length cutoff l and minimal score cutoff m. Formally, we denote by $X(l, m)$ a random variable for the length of an index list in index $I(l, m)$, and want to estimate the distribution $F(l, m)$ of that random variable, i.e., estimate $F(l, m; x) = P[X(l, m) \leq x]$. We sample the index lists for a subset P of n keys chosen independently from all keys; each sample yields a value $X_i(l, m)$ for the length of that list in $I(l,m)$. Using the empirical distribution function [Was05], we can estimate the cdf of this distribution as

$$\hat{F}_n(l,m;x) := \frac{\sum_{i=1}^{n} J(X_i(l,m) \leq x)}{n}, \text{ where}$$

$$J(X_i(l,m) \leq x) = \begin{cases} 1 & \text{if } X_i(l,m) \leq x \\ 0 & \text{else} \end{cases} .$$

All we actually need is the expected length $E[F(l,m)]$, which can again be estimated from the sample as $\overline{X_i(l,m)}$ [Was05]. Assuming that there are $K(P)$ keys in the sample, the expected number of entries in the index for the sample is therefore $K(P) \cdot \overline{X_i(l,m)}$. To extend this estimate to the complete collection, we make sure that the size of P relative to the size of the collection is known, for example by sampling $p\%$ of all keys (this can be easily implemented using hash values of keys). The expected number of keys in the index is therefore $\frac{100 \cdot K(P)}{p}$, and the expected number of entries in the index is

$$N(l,p) := \frac{100 \cdot K(P)}{p} \cdot \overline{X_i(l,m)}.$$

The size estimator for a compressed index is built similarly, but instead of computing just the length $X_i(l,m)$, we materialize and compress the list, and use its actual size, avoiding the need to estimate the average value of s.

As the space of feasible values for the parameters l and m is in principle infinitely large, we cannot compute the estimate for all combinations. Instead, our implementation considers only selected step sizes for l and m, computes estimates for those values, and interpolates sizes for other value combinations. We currently consider a step size of 100 for l and 0.05 for m.

Index Quality

Intuitively, the fewer entries we keep in each list, the more will reduce the quality of query results, since the probability that relevant documents are dropped from the pruned lists increases. The goal is to find values for $\overline{m}$ and $\overline{l}$ that *maximize index quality* while generating an index that fits into a predefined amount of memory. We now define different notions of *index quality* measures $M(C,l,m,k)$ for index $I(C,l,m)$ and a fixed number k of results.

In the best case, a set of predefined *reference* or *training topics* Λ is available that include human assessments of the relevance of documents in the collection. Such a set of topics can be build, for example, by first selecting a set of representative topics from a query log, then computing top-k results for different parameter settings, pooling those results per topic, and have human assessors determine the relevance of each result. Alternatively, click logs could be used to estimate the relevance of results (but with much lower confidence). Topic sets of this kind are frequently available for test collections such as TREC .GOV or .GOV2, but they cannot be reused for different document collections. Given such a set Λ of reference topics, we denote by $p_\Lambda[k;I]$

the average quality of the top-k results over all topics (e.g., precision@k or NDCG@k) computed using index I; our implementation currently uses average precision at k. We can now define *effectiveness-oriented* and *efficiency-oriented absolute index quality*:

- *Effectiveness-oriented absolute index quality*: this is quantified as the ratio of the quality of the first k results with the pruned index to the quality of the first k results with the unpruned index or, formally, $\frac{p_\Lambda[k;I(C,l,m)]}{p_\Lambda[k;I(C)]}$.
- *Efficiency-oriented absolute index quality*: this is quantified as the reciprocal of the maximal query processing cost per query term and query term pair (i.e., $\frac{1}{l}$) when the result quality of the pruned index is not worse than that of an unpruned term-only index without proximity lists (formally, when $\frac{p_\Lambda[k;I(C,l,m)]}{p_\Lambda[k;T(C)]} \geq 1$), and 0 otherwise.

Here, the effectiveness-oriented index quality measure aims at finding the best possible results by including as much proximity information in the index as possible. The efficiency-oriented quality measure, on the other hand, assumes that the quality of a term-only index is already sufficient and tries to minimize the length of index lists (assuming that query processing efforts are directly proportional to the lengths of index lists).

For most applications, such a set of reference topics does not exist or would be too expensive to generate. In this case, we fix a set Γ of *queries* (e.g., representative samples from a query log) and use *relative quality* to estimate how good results with the pruned index are, compared to results with the unpruned index. We define, for each query $\gamma_i \in \Gamma$, the set of relevant results to be the top-k documents with some index configuration I' and use this to compute the result quality of index configuration I. When the quality measure is precision, this boils down to computing the overlap of the top-k results with index configurations I and I'. We formally denote the resulting quality of index I as $p_\Gamma[k;I|I']$.

We can now define relative index quality measures in an analogous way to the absolute measures defined before. However, we then would always favor index configurations that produce exactly the results of the corresponding unpruned index, as we assume that any results not in the top-k results with the unpruned index are non-relevant. This is often overly conservative in practice, as many of the new results will be relevant to the user as well, so it is usually sufficient to provide a "high" overlap, not a perfect one. We therefore introduce another application-specific tuning parameter α that denotes the threshold for relative quality above which we accept an index configuration. This is especially important for efficiency-oriented index quality: we cannot expect that we will get the same results with the pruned index with term and combined lists as with just the unpruned term lists, so achieving an overlap of 1 there would be impossible. Instead, we use $I(C)$ also in that case and set α to a value below 1.

- *Effectiveness-oriented relative index quality*: this is the relative result quality of the pruned index $p_\Gamma[k;I(C,l,m)|I(C)]$.

- *Efficiency-oriented relative index quality*: this is the reciprocal of the maximal query processing cost per query term and query term pair (i.e., $\frac{1}{l}$) when the relative result quality of the pruned index $p_\Gamma[k; I(C,l,m)|I(C)]$ is at least α and 0 otherwise.

Index Tuning

We can now formally specify the index tuning problem:

Problem 1. *Given a collection C of documents, an upper limit S for the index size, a target number of results k, and an index quality measure M, estimate parameters $\overline{m}$ and $\overline{l}$ such that $M(C, \overline{l}, \overline{m}, k)$ is maximized, under the constraint that $|I(C, \overline{l}, \overline{m})| \leq S$. When there is more than one combination of m and l that maximize the quality measure and satisfy the size constraint, pick one of them where the index size is minimal.*

Note that even though the index is tuned for a specific number k of results, it can be still used to retrieve any other number of results. We will experimentally validate in Section 8.5.2 that result quality does not degrade much in these cases.

8.3.2 Implementation of the Tuning Framework

We implemented our tuning framework within the MapReduce paradigm [DG08], dividing the tuning process into several map-reduce operations. As stated before, the input to the tuning process is the collection C, a target index size S, a target number of results k, and an index quality measure M that includes a set of training topics T. Additionally, we fix the fraction p of index keys (for both terms and term pairs) to be sampled. The tuning process then proceeds in the following order, where each step is implemented as a map-reduce operation:

1. **Compute index for sample and training topics.** The *map* phase considers each document in the collection, parses it, and creates index entries for terms and term pairs that are either part of the sample or the training topics. These entries are still incomplete, because the final BM25 scores can be computed only when global properties of the collection are known, so they contain only term frequencies and document lengths (but already complete $acc_d(t_1, t_2)$ values for term pairs); their key is the term or term pair. The *reduce* phase then combines items with the same key into an index list, completing their scores as all global parameters of the score (average document length, number of documents, and document frequency of each term) are now known[1]. The output of this phase are two indexes, one for the sample, the other for the set of training topics.

[1] At least Hadoop 0.20 does not directly provide these global parameters to the reduce phase, so we need to store them in files and aggregate them in each reducer. The alternative would be to combine the initial map with a do-nothing reducer, include additional map-reduce operations to compute the global values, and then have a map-reduce operation with a do-nothing mapper and the reducer we just described.

2. **Prepare the estimator for the index size.** The *map* phase considers each key in the sample and computes, for each combination (l, m) it considers, the size s of the corresponding index list when pruned according to the l and m cutoffs (or the size of its compressed representation for compressed indexes), which is then written out with key (l, m). The algorithm starts with $l = k$ and increases it by the step size for l, and considers all values for m, starting at 0 and increasing it by the step size for m. The *reduce* phase combines all values for a single pair of (l, m) cutoffs and computes the average index list size for this cutoff. This value is then stored in an on-disk data structure as size estimate for (l, m). This phase also counts the overall number of keys in the sample.

3. **Prepare solving the optimization problem.** In an initial map-reduce operation, we compute the baseline precisions. The *map* phase then considers each topic with its corresponding assessments and computes, for each (l, m) pair provided by the size estimator, the quality of the index for this topic. This can be efficiently implemented by a stepwise incremental join algorithm.

   In the first step, the algorithm sets $l = k$, i.e., it reads the first k entries from each list and incrementally computes results for (k, m), starting at the highest value for m and decreasing it by the step size of m. This yields, for each m, a temporary set of results with (partial) scores, from which the k documents with highest partial score are considered as result. The index quality for this result is computed and written out with key (k, m). If the score of the entry at position k is less than m (i.e., the list would be cut before it), the value m is marked as completed and will not be considered later. As soon as m exceeds the score of the last read entry, all smaller values for m will get the same index quality.

   In the following steps, the algorithm reads more entries from each list corresponding to the step size for l. Assume that it read up to l entries from each list. It continues with the temporary set of partial results from the previous step and the highest value for m not yet marked as completed and repeats the above process. This phase ends when either all values for l have been considered or all lists have been completely read. It is evident that each entry of the lists is read at most once, so the complexity is linear in the aggregated number of entries in the index lists for this topic.

   Note that for the efficiency-oriented quality measures, the map phase does not write the actual index quality measure introduced in Section 8.3.1: instead of $1/l$, the reciprocal of the maximal query processing cost per query term and query term pair, respectively, the map phase writes the actual precision of the top-k results. This is due to the fact that only (l, m) combinations are valid that can provide a given precision (averaged over all training topics in T). The reduce phase will transfer the precision values for each valid (l, m) combination to such an index quality measure later.

   The *reduce* phase averages, for each combination of (l, m), the per-topic index quality values computed by the map phase, and computes the final index quality for this combination. For the efficiency-oriented measures, this means that it compares the average precision with the result quality of the term-only index and uses $1/l$ as final index quality when the average precision is high enough. If the (l, m) combination has a non-zero index quality, the reducer estimates its size using the size estimator. For each (l, m) combination with a non-zero index quality that matches the size constraint S, the reduce phase outputs an (l, m, q, s) tuple, where q is the index quality and s is the index size.

4. **Compute an approximate solution of the optimization problem.** The following centralized phase scans all output tuples from the previous step and determines the tuple $(\overline{l}, \overline{m}, \overline{q}, \overline{s})$ with highest quality. Optionally, it can further explore the solution space around $(\overline{l}, \overline{m})$ for better solutions. The output of this step is an approximate solution to Problem 1.

5. **Materialize the final index.** Analogously to phase 1, the final index is materialized in a single map-reduce operation. Note that each mapper can already restrict the index entries it generates: for term pair entries, it does not emit any entries whose score is below $\overline{m}$, and for term entries, it emits only the $\overline{l}$ entries with highest scores (which can be achieved using an additional combiner). An additional optimization for this step would be to generate only an approximation of the final index: if there are M mappers used to parse the collection, each mapper needs to emit at most $\frac{\beta}{M} \cdot \overline{l}$ entries, where $\beta \geq 1$ is a tuning parameter that steers the expected number of entries missing in the final index.

8.4 Log-Based Term Pair Pruning

Even with relatively short list length cutoffs $\overline{l}$, the overall space consumption of the pruned combined lists can still be pretty huge, because there are a lot more combined lists than term lists. On the other hand, the majority of combined lists are unlikely to ever occur in any query. A possible solution can be to selectively materialize only combined lists for term pairs that occur at least t times in a query log, which can drastically reduce the number of lists. When counting term pairs in the query log, we consider each query separately, build up all possible term pair combinations for that particular query, and finally count for each term pair the number of occurrences over the complete query log. However, when one of these unlikely queries is issued for which not all or even no combined lists are available, answering it using the pruned term lists and the available subset of combined lists only may affect the result quality for this query. Figure 8.5 demonstrates this effect, using the AOL query log and our training topics on .GOV2 (see Section 8.5.1), with $\overline{l} = 4310$ and $\overline{m} = 0.00$. The x-axis of this chart shows different values for the threshold t of term pairs in the AOL log, and the y-axis shows the precision at 10 results. The line with diamonds depicts the result of running our merge-based algorithm from Section 7.3.5 with the available index lists only. It is evident that the higher the threshold, the lower the result quality gets, which can be explained by fewer and fewer combined lists being materialized. For very high thresholds (not depicted in the chart), the precision drops to 0.396, compared to 0.617 when using all lists.

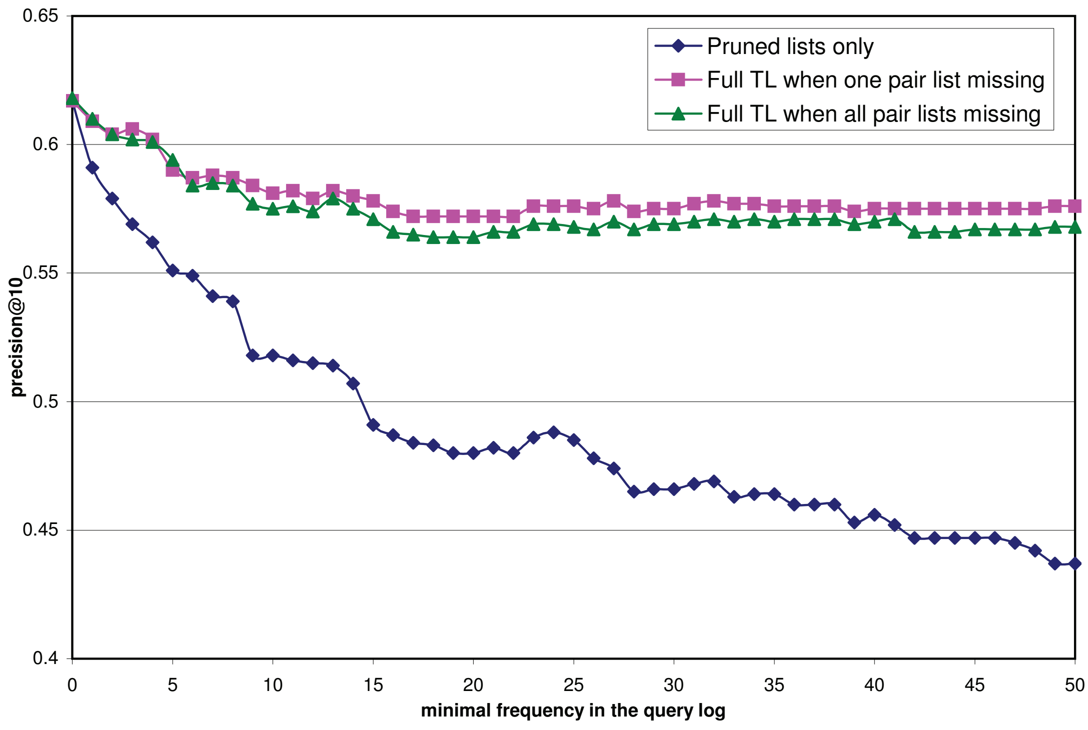


Figure 8.5: Effect of log-based pruning on query performance (on training topics).

To overcome this negative effect, we propose to keep the unpruned term index lists when log-based pruning is applied. As soon as at least one combined list for a query term is missing (variant 1) or, alternatively, all combined lists (variant 2) for a query term are missing, we read the available combined lists and the unpruned term list for that term.

This improves result quality to at least the quality of an unpruned term index, but at the same time incurs an increased cost for query evaluation as longer term lists have to be read. Figure 8.5 also depicts the effect of these approaches on result quality (line with squares: read full term lists when at least one pair is missing; line with triangles: read full term lists when all combined lists are missing). It is evident that this combined execution helps to keep precision close to the level of the precision with the unpruned $T(C)$ index only (which is 0.585). Our tuning framework can be extended to consider only combined lists where the corresponding term pair occurs at least t times in a query log, and tunes the parameters to reach the optimization goals even with this limited selection of combined lists. Maintaining only the term pair lists for term pairs that appear in previous queries (e.g., term pairs that appear in queries from a query log) may be restrictive for rare queries that will appear in the future. For those few queries which are affected, the results of the proposed approach would not benefit from term proximity. However, we can still achieve a retrieval quality similar to that using BM25 scores as we can use the unpruned term lists. If rare queries become more frequent over time, one may consider using an updated query log file to update the index structures.

To accelerate query processing and to save on accumulators, we split unpruned term lists in two pieces: the $\bar{l}$ entries with highest scores are stored in docid order and the remaining entries in score order. When processing a query where some combined lists are missing, in a first phase we process the first piece of the term lists and the available combined lists with the merge-based algorithm from Section 7.3.5, keeping all documents and their scores in memory. After that, in a second phase, a standard top-k algorithm (in our case NRA, cf. Section 6.1) consumes the second piece of the term lists, using the already read documents as candidates. The acc_d contribution for non-available combined lists is 0 in both steps. This algorithm will terminate more quickly than running it on the unpruned term lists alone, and will usually give better results due to the proximity score from the combined lists.

We give a more detailed explanation of how we process queries for the case of a 4-term-query $\{t_1, t_2, t_3, t_4\}$ where one pruned combined list is missing (due to non-sufficient frequency of that term pair in the query log). W.l.o.g. assume that the missing combined list is CL(t_1, t_2). For both variants, we load pruned CLs for (t_1, t_3), (t_1, t_4), (t_2, t_3), (t_2, t_4), and (t_3, t_4). For variant 1, we load the pruned TLs for t_3 and t_4, the unpruned TL for t_1 and for t_2 as at least one combined list for t_1 and at least one pruned combined list for t_2 is missing, namely (t_1, t_2). In the first phase, we process the pruned TLs, the first piece (docid-ordered) of the unpruned TL for (t_1, t_2), and the available pruned CLs using an n-way merge join algorithm. In the second phase, we process the second, score-ordered piece of the unpruned TL for (t_1, t_2) using NRA with the already seen documents as candidates. For variant 2, we load only pruned TLs for all query terms as for every query term not all combined lists are missing. As we work with pruned lists only, we only execute an n-way merge join on the pruned lists, the second phase is not needed.

8.5 Experimental Evaluation

This section presents results of a large-scale experimental evaluation of our techniques with two standard text collections. To facilitate reading, we first give a short overview of the content and the goals for each subsection. Section 8.5.1 gives details about the experimental setup and the employed test beds. Section 8.5.2 describes the evaluation of our index tuning techniques from Section 8.3 for different result size cardinalities (10 and 100). Both for effectiveness- and efficiency-oriented index quality measures, we present parameter tuning results given a size limit for the pruned indexes. We present results that tune indexes in the presence (absolute index quality) and absence (relative index quality) of relevance assessments. The goal of Section 8.5.2 is to show the feasibility of our index tuning techniques. Section 8.5.3 compares the query processing performance of merge joins with pruned indexes as input to the recently proposed Block-Max WAND (BMW) algorithm [DS11], a state-of-the-art document-at-a-time algorithm that uses dynamic pruning on unpruned indexes. Besides the original BMW, we provide additional results for our proximity score-enhanced BMW variant. The goal of Section 8.5.3 is to compare the query performance of merge joins with pruned indexes to dynamic pruning with unpruned indexes. The query processing performance is measured both by abstract measures (e.g., the number of opened lists, average number of entries and bytes read from disks) and average query processing times for hot and cold cache settings. A running system (i.e., a warm cache scenario) is simulated using an LRU-based cache. For various cache sizes, we report cache hit ratios, the number of non-cached lists, the warm cache query processing times, and for BMW, in addition, the number of read blocks. Section 8.5.4 evaluates the effect of query log-based combined list pruning, one way to shrink the index size that is orthogonal to index compression techniques. Section 8.5.5 summarizes the conclusions from Section 8.5.2 to Section 8.5.4. Section 8.5.6 presents additional results with ClueWeb09 and aims at demonstrating the scalability of our index tuning approach by means of similar experiments as the ones shown in Section 8.5.2. Section 8.5.7 describes our efforts that apply our tuning framework to the INEX 2009 test bed for the Efficiency Track.

8.5.1 Setup

We evaluated our methods with two standard text collections from TREC[2], the GOV2 collection and the ClueWeb09 collection (cf. Section 3.2.1 for more details about both collections). The TREC GOV2 collection consists of approximately 25 million documents from U.S. governmental Web sites with an uncompressed size of approximately 426GB. We used the 100 Ad Hoc Task topics from the TREC 2004 and 2005 Terabyte Tracks[3] (cf. Appendix B, Tables B.1 and B.2) as training topics for tuning index parameters, and the 50 Ad Hoc Task topics from the TREC 2006 Terabyte Track (cf. Appendix B, Table B.3) for testing the quality of results. We used the AOL query log[4] for the log-based technique. We measure result quality as precision values P@k, i.e., the average number of relevant results among the first k results and additionally report normalized discounted cumulative gain (NDCG@k) [JK02] that considers the order of results, not the result set as a whole.

The ClueWeb09 collection[5] consists of approximately 1 billion Web documents crawled in January and February 2009. Following standards at the TREC Web Track, we consider only the approximately 500 million English documents (e.g., also used by [NC10]), from which we chose the 50% documents with the smallest probabilities to be spam according to the Waterloo Fusion spam ranking[6] (spaminess has also been

[2]`http://trec.nist.gov`
[3]`http://trec.nist.gov/data/terabyte.html`
[4]`http://gregsadetsky.com/aol-data/`
[5]`http://boston.lti.cs.cmu.edu/Data/clueweb09/`
[6]`http://durum0.uwaterloo.ca/clueweb09spam/`

used in [BFC10] for example). The resulting document set has an uncompressed size of about 6TB. We use the 50 topics from the Ad Hoc Task of TREC Web Track 2009[7] (cf. Appendix B, Table B.4) to train and optimize the index parameters; documents without assessment are considered non-relevant. The 50 topics from the Ad Hoc Task of Web Track 2010[8] (cf. Appendix B, Table B.5, non-assessed topics are marked) are employed as test topics. Due to a few missing relevance assessments for the test topics, we only provide precision values for our runs submitted to Web Track 2010, based on a subset of 48 topics with assessments as published in our contribution to TREC 2010 [BS10].

The most significant part of our experiments is built on the GOV2 collection, since the number of available topics for the ClueWeb09 collection is lower and the assessments sparser than for the GOV2 counterpart. Therefore, we run only a limited set of experiments on the ClueWeb09 collection and report detailed tuning results for GOV2 only.

As an additional test bed we employ the test bed from INEX 2009. The INEX Wikipedia collection from 2009 consists of approximately 2.67 million articles and 1.4 billion elements. There are two types of queries used in our experiments: 115 Type A topics from the Ad Hoc Track which includes classic Ad Hoc-style focused passage or element retrieval with a combination of NEXI CO and CAS queries. They are partially enriched with phrasetitle elements that indicate important phrases in the title field. 115 Type B topics have been generated from the Type A topics by running Rocchio-based blind feedback on the results of the article-only Ad Hoc reference run. Therefore, Type B topics can consist of partly more than 100 keywords. We show results from our participation in the INEX 2009 Ad Hoc and Efficiency Tracks where we evaluate CO queries (i.e., title fields and partially phrasetitle fields). Type A topics are listed in Appendix C, Table C.5 to C.8, Type B topics are not listed due to their length.

Whenever we report times for parameter tuning or index construction, they were measured on a cluster of 10 servers in the same network, where each server had 8 CPU cores plus 8 virtual cores through hyperthreading, 32GB of memory, and four local hard drives of 1TB each. The cluster was running Hadoop 0.20 on Linux, with replication level set to two. Query execution times are reported using a single core of a CPU of a single node in the cluster. All algorithms are implemented in Java 1.6.

We distinguish between test indexes and full indexes: while test indexes contain only lists for terms that are part of topics in a given test bed and term pairs that can be built from each topic, full indexes do not make this restriction, but contain lists for all terms and term pairs (occurring in a window of size W=10). If not explicitly stated differently, we build up test indexes to keep the indexing effort manageable as to both the time required for index construction and the required space on disk. Hence, we can maintain many indexes for evaluation on the same disk.

[7]`http://trec.nist.gov/data/web09.html`
[8]`http://trec.nist.gov/data/web10.html`

8.5.2 Index Tuning on GOV2

We evaluated our index tuning techniques from Section 8.3 for different maximal index sizes and result counts (10 and 100). We report all results in this section with and without index compression. We are aware of the fact that compression is preferable to using uncompressed indexes, because the data can be read faster and decompressing is less expensive than reading more data. We do not exclude experiments without compression from this section as we want to be able to quantify the effects of compression within our tuning framework. The effect of log-based combined list pruning will be evaluated in Section 8.5.4. For each setting, we first estimated index parameters using the training topics, built an index with these parameters, and then evaluated result quality on the test topics.

Absolute Index Quality

Table 8.1 shows the results of index tuning on the training topics with selected size limits below the collection size, for uncompressed indexes. In this table, each row shows results for a given index size constraint and number of query results, namely the resulting index parameters, the estimated and real index size for these parameters, and the result quality on the training and test topics with this index. The rows with size limit ∞ denote the corresponding unpruned indexes with term+combined lists (named $I(C)$) or term lists (named $T(C)$), respectively. To build up the unpruned combined lists, we consider only term pair occurrences within a text window of fixed size $W = 10$ as used in Section 7.3. Estimating one set of parameters took approximately 5 hours, where about 3.5 hours were required for the first map-reduce phase to build the index for the sample and the training topics. The time for building the final index strongly depends on the chosen parameters; for a full index with up to 310 entries per list and a score threshold of 0.05, this took less than five hours on our cluster.

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	P@k on train	P@k on test	NDCG@k on train	NDCG@k on test
effective-ness oriented index quality	10	100GB	19010	0.40	96.4	96.9	0.596	0.572	0.4766	0.5121
		200GB	19010	0.15	170.5	170.8	0.610	0.578	0.4819	0.5153
		400GB	4310	0.00	396.4	396.4	0.617	0.592	0.4874	0.5262
		∞	$I(C)$			757.0	0.614	0.578	0.4875	0.5158
	100	100GB	10200	0.35	97.4	97.6	0.3899	0.3146	0.4117	0.4174
		200GB	17900	0.15	169.1	169.4	0.3975	0.3244	0.4190	0.4256
		400GB	4200	0.00	394.3	394.3	0.4035	0.3176	0.4245	0.4204
		∞	$I(C)$			757.0	0.4108	0.3338	0.4264	0.4324
efficiency-oriented index quality	10	100GB	5010	0.30	87.2	87.0	0.586	0.578	0.4634	0.5174
		200GB	310	0.05	128.1	127.9	0.588	0.534	0.4658	0.4732
		∞	$T(C)$			22.9	0.585	0.538	0.4701	0.4852
	100	400GB	800	0.00	270.6	270.6	0.3848	0.2850	0.4093	0.3888
		∞	$T(C)$			22.9	0.3847	0.3002	0.4078	0.3954

Table 8.1: GOV2: index tuning results for absolute index quality without index compression.

It is evident that all indexes with the estimated parameters meet the index size constraint. For the effectiveness-oriented quality goal, all precision results (for the training and, more importantly, also for the test topics) are better than the precision with an unpruned term-only index (significantly better under a paired t-test with $p < 0.05$ when the size limit is at least 200GB), so the additional combined index lists help to improve precision even when they are pruned. For the efficiency-oriented quality goal, it turns out that already very short list prefixes (310 entries for top-10, 800 entries for top-100 results) are enough to yield results with a quality comparable to standard term indexes, given a sufficiently large index size constraint. If this constraint is too tight, short lists cannot guarantee the quality target.

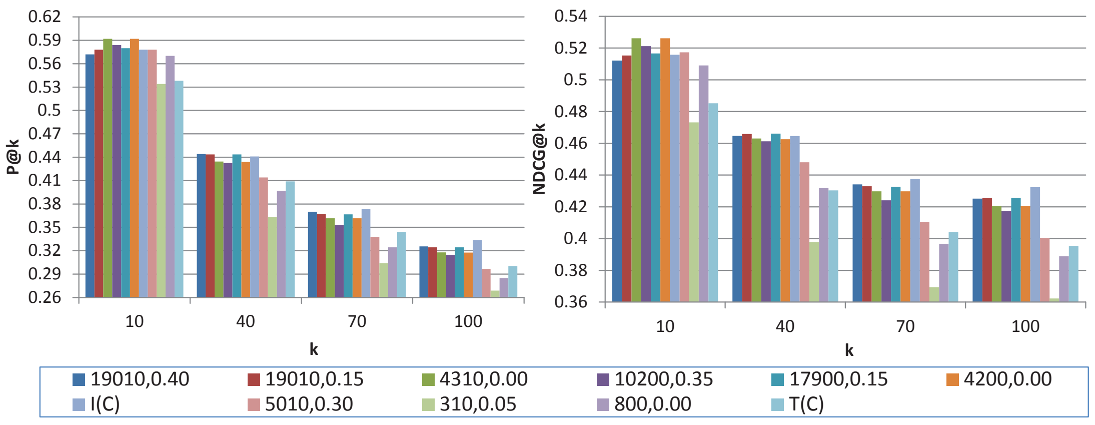


Figure 8.6: P@k and NDCG@k on test topics for effectiveness- and efficiency-oriented absolute index quality without index compression.

Although we tune for document retrieval of either the best k=10 or k=100 result documents, we are aware that sometimes it may be necessary to retrieve a number of results k' that is different from the number of results k used for tuning. Figure 8.6 shows P@k and NDCG@k values for the test topics with all index configurations from Table 8.1 for varying numbers of retrieved results, namely 10, 40, 70, and 100 result documents. It is evident that result quality with indexes tuned for k=10 results does not degrade much when returning longer result lists, i.e., choosing k' greater than 10. Differences for NDCG are a bit larger than for precision as we tuned our indexes using the precision measure. Compared to the original runs from the TREC 2006 Terabyte Track [BCS06], our tuned indexes do well in terms of precision. The best P@20 we get for the effectiveness-oriented goal is 0.5310 (for (10200, 0.35)), none of the P@20 values underscores 0.5210. Our best indexes outperform 14 of 20 competitors in P@20. Note that our index tuning was not carried out with the TREC 2006 topics but with the training topics and for retrieval of the top-10 or top-100 results instead of the top-20, which imposes a penalty on us. For the efficiency-oriented goal the best index (5010, 0.30) reaches a P@20 of 0.5210; very short list lengths deteriorate in later precision values, at 0.4520 for (310, 0.05).

Table 8.2, which has the same layout as the table before, shows the results of index tuning on the training topics with selected size limits below the collection size for compressed indexes. Our index compression scheme is effective: an index in configuration (4310, 0.00) requires 396.4GB uncompressed, but only 248.8GB compressed, an index in configuration (5010, 0.30) requires 87.0GB uncompressed, but only 55.4GB compressed, and an index in configuration (310, 0.05) requires 127.9GB uncompressed, but only 94.9GB compressed.

Opt.		size			size[GB]		P@k on		NDCG@k on	
goal	k	limit	$\bar{l}$	$\overline{m}$	est.	real	train	test	train	test
effectiveness oriented index quality	10	50GB	6810	0.75	48.3	48.1	0.589	0.586	0.4612	0.5144
		70GB	19010	0.30	64.6	64.5	0.599	0.574	0.4779	0.5140
		100GB	19010	0.20	98.5	98.0	0.608	0.574	0.4795	0.5139
		200GB	19010	0.05	173.8	172.8	0.615	0.584	0.4861	0.5182
		400GB	4310	0.00	249.7	248.8	0.617	0.592	0.4874	0.5262
		∞	$I(C)$			468.9	0.614	0.578	0.4875	0.5158
	100	50GB	10400	0.85	49.9	49.7	0.3795	0.3124	0.4017	0.4120
		70GB	19800	0.30	64.9	64.7	0.3926	0.3250	0.4156	0.4249
		100GB	20000	0.20	99.0	98.5	0.3970	0.3248	0.4189	0.4260
		200GB	19700	0.05	174.4	173.4	0.4008	0.3264	0.4223	0.4273
		400GB	14400	0.00	295.1	293.5	0.4067	0.3272	0.4263	0.4288
		∞	$I(C)$			468.9	0.4108	0.3338	0.4264	0.4324
efficiency-oriented index quality	10	50GB	6310	0.75	47.9	47.7	0.585	0.588	0.4591	0.5156
		70GB	5010	0.30	55.6	55.4	0.586	0.578	0.4633	0.5173
		100GB	310	0.05	94.9	94.9	0.588	0.534	0.4658	0.4732
		200GB	310	0.05	94.9	94.9	0.588	0.534	0.4658	0.4732
		400GB	310	0.05	94.9	94.9	0.588	0.534	0.4658	0.4732
		∞	$T(C)$			14.5	0.585	0.538	0.4701	0.4852
	100	50GB	10400	0.85	49.9	49.7	0.3795	0.3124	0.4017	0.4120
		70GB	6000	0.30	56.9	56.7	0.3847	0.2988	0.4077	0.4026
		100GB	3600	0.15	84.6	84.2	0.3849	0.2946	0.4099	0.3979
		200GB	900	0.00	193.8	193.5	0.3861	0.2868	0.4108	0.3902
		400GB	900	0.00	193.8	193.5	0.3861	0.2868	0.4108	0.3902
		∞	$T(C)$			14.5	0.3847	0.3002	0.4078	0.3954

Table 8.2: GOV2: index tuning results for absolute index quality with index compression.

It is evident that all indexes with the estimated parameters meet the index size constraint, and the size estimator only slightly overestimates the final index size. For the effectiveness-oriented quality goal, all precision results for indexes with a size constraint of at least 70GB (for the training and, more importantly, also for the test topics) are better than the precision with an unpruned term-only index (significantly better under a paired t-test with $p < 0.05$ when the size limit is at least 200GB), so the additional combined index lists help to improve precision even when they are pruned. NDCG results behave similarly. For the efficiency-oriented quality goal, it turns out that already very short list prefixes (310 entries for top-10, 900 entries for top-100 results) are enough to yield results with a quality comparable to standard term indexes, given a sufficiently large index size constraint. If this constraint is too tight, short lists cannot guarantee the quality target.

Note that index tuning for different index size limits may result in identical optimal index parameters $(\bar{l}, \overline{m})$; for example, efficiency-oriented index quality tuning for top-

10 retrieval and size limits between 100GB and 400GB results in (310,0.05). Recall from the description of efficiency-oriented index quality and Problem 1 in Section 8.3 that index list pruning aims at minimizing the list length l and as an afterthought the index size; anyway, the respective pruned index has to provide at least the P@10 quality of unpruned term lists. As (l, m) combinations with $l < 310$ or $l = 310$ and $m > 0.05$ cannot provide the same P@10 values that unpruned term lists provide, the optimal index parameters (310,0.05) remain constant. In this case, the resulting pruned index does not use the full amount of space given by the size limit, which comes as a consequence of the efficiency-oriented quality definition. Efficiency-oriented tuning aims at minimizing the maximum query processing cost per list and provides at least the same retrieval quality as using unpruned $T(C)$ indexes. If the retrieval quality goal can be met with small indexes, we do not waste space.

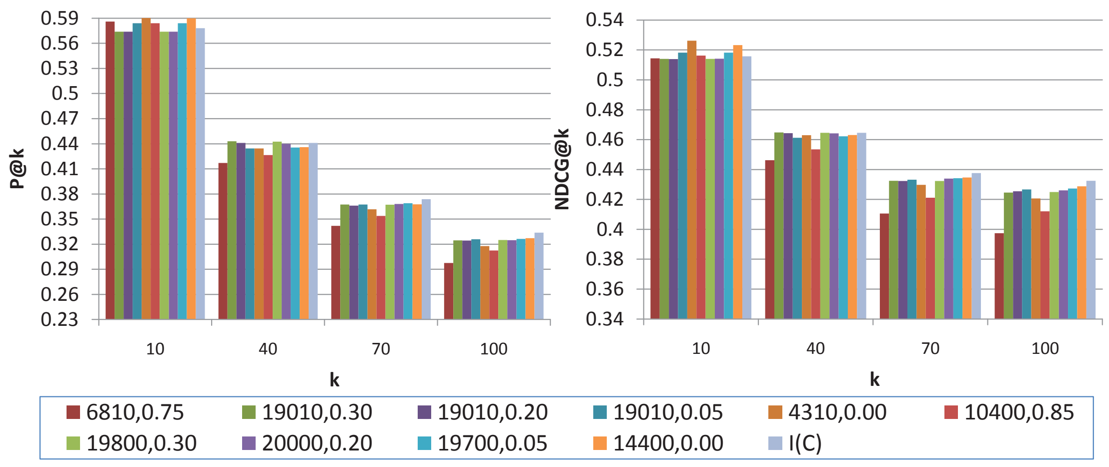


Figure 8.7: P@k and NDCG@k on test topics for effectiveness-oriented absolute index quality with index compression.

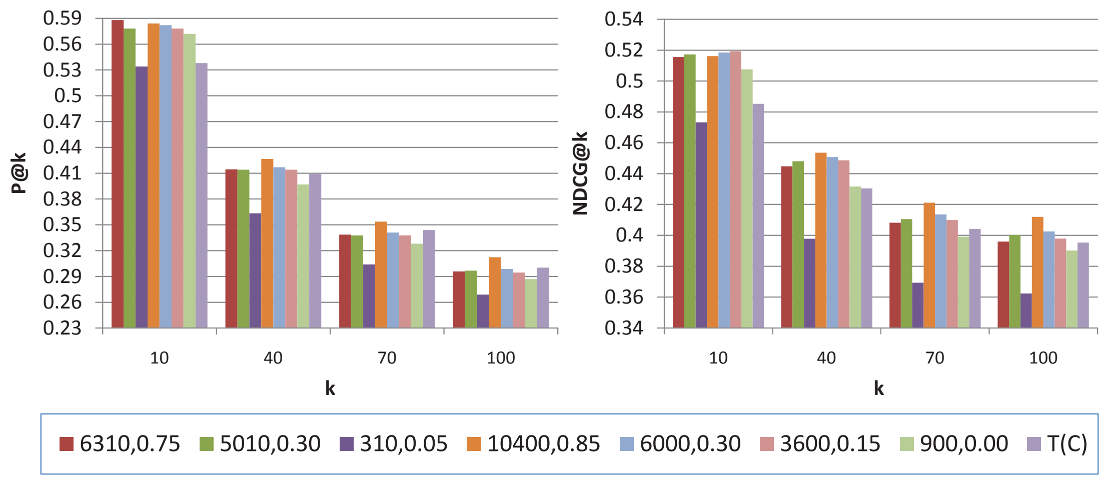


Figure 8.8: P@k and NDCG@k on test topics for efficiency-oriented absolute index quality with index compression.

Although we tune for retrieving either the best k=10 or k=100 result documents, it can often happen that a different number of results should be retrieved. Figures 8.7 and 8.8 show precision and NDCG values for the test topics with all index configurations from Table 8.2 for varying numbers of retrieved results, namely 10, 40, 70, and 100 result documents. In Figure 8.7, for each choice of k, the rightmost bar represents $I(C)$, the baseline for effectiveness-oriented tuning; in Figure 8.8, the rightmost bar for each k represents $T(C)$, the baseline for efficiency-oriented tuning. It is evident that result quality with pruned indexes tuned for k=10 results does not degrade much relative to the result quality provided by $T(C)$ (the baseline for efficiency-oriented tuning) or $I(C)$ (the baseline for effectiveness-oriented tuning) when returning more results. Even if we select the weakest setting at late precision values, namely (310,0.05), we still achieve a P@100 value of 0.26 compared to 0.30 for $T(C)$. Differences for NDCG are slightly larger, which could be expected since we tuned for precision, not NDCG.

Relative Index Quality

We first performed an experiment to estimate good values for α, the application-specific tuning parameter that denotes the threshold for relative quality above which we accept an index configuration: we computed, for a selection of possible values for α, optimal index parameters $(\bar{l}, \overline{m})$ for the training topics under relative index quality, then instantiated the corresponding pruned indexes and compared the resulting absolute precisions (using the assessments from TREC) to the precision of the same topics with $I(C)$ and $T(C)$. The results of this experiment are displayed in Table 8.3. This allows to estimate values for α that are sufficient to yield similar precision values as the unpruned term-only index for the efficiency-oriented measure. A good choice for α is 0.75 as $\frac{p[100;I(C,l,m)]}{p[100;T(C)]}$ is close to 1 which means that, using pruned term and pruned combined lists for top-100 document retrieval, we achieve a precision comparable to that using unpruned term lists.

α	0.7	0.75	0.8	0.85	0.9	0.95
$\frac{p[100;I(C,l,m)]}{p[100;I(C)]}$	0.9343	0.9471	0.9626	0.9759	0.9914	1.0010
$\frac{p[100;I(C,l,m)]}{p[100;T(C)]}$	0.9945	1.0081	1.0246	1.0388	1.0553	1.0655

Table 8.3: Relative result quality for different values of α.

Table 8.4 gives tuning results for relative index quality with uncompressed indexes. We can get close to the result quality for top-10 results of an unpruned index with the effectiveness-oriented techniques (we even get better quality for some scenarios), for both the test and the training topics. For top-100 results, the situation is slightly worse, there is a small gap to the quality of an unpruned index (which, however, may be tolerable). For the efficiency-oriented indexes, we achieve comparable or even better precision values than for the unpruned text indexes, at a reasonable index size of less than 100GB. Figure 8.9 depicts P@k and NDCG@k values for efficiency-oriented and effectiveness-oriented index quality goals on all (l, m) combinations from Table 8.4,

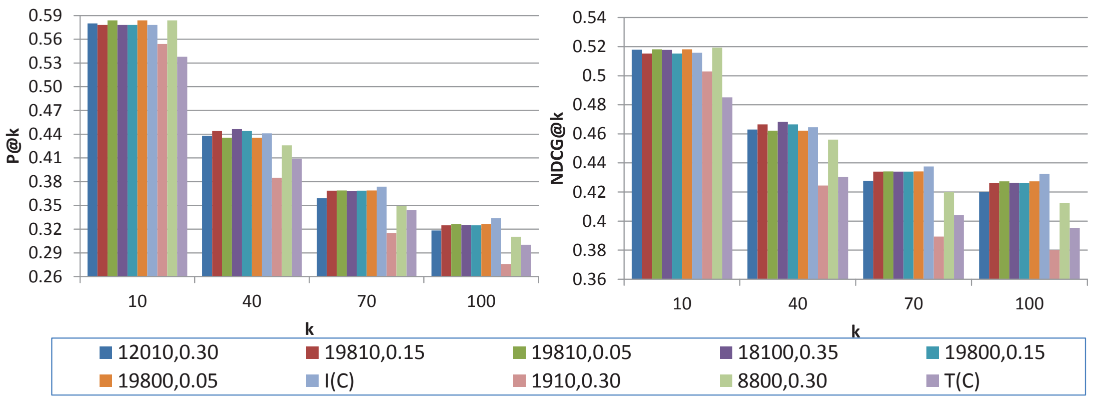


Figure 8.9: P@k and NDCG@k on test topics for effectiveness- and efficiency-oriented relative index quality without index compression.

for varying numbers of retrieved results. It is evident that the relative index quality approach ensures retrieval quality on test topics even without relevance assessments. Like stated before, the result quality of indexes tuned for k=10 results does not degrade much for more retrieved results relative to $T(C)$ and $I(C)$, respectively and for both retrieval measures.

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	overlap on train	P@k on train	P@k on test	NDCG@k on train	NDCG@k on test
effective-ness oriented index quality	10	100GB	12010	0.30	99.7	100.1	0.837	0.591	0.580	0.4711	0.5179
		200GB	19810	0.15	171.4	171.8	0.893	0.609	0.578	0.4809	0.5153
		400GB	19810	0.05	293.3	293.4	0.924	0.615	0.584	0.4858	0.5182
		∞	$I(C)$			757.0	-	0.614	0.578	0.4875	0.5158
	100	100GB	18100	0.35	100.0	100.4	0.773	0.3899	0.3252	0.4120	0.4263
		200GB	19800	0.15	171.4	171.8	0.829	0.3983	0.3248	0.4201	0.4260
		400GB	19800	0.05	293.3	293.4	0.868	0.4008	0.3266	0.4222	0.4274
		∞	$I(C)$			757.0	-	0.4108	0.3338	0.4264	0.4324
efficiency-oriented index quality	10	100GB	1910	0.30	73.1	72.7	0.750	0.574	0.554	0.4549	0.5030
		∞	$T(C)$			22.9	-	0.585	0.538	0.4701	0.4852
	100	100GB	8800	0.30	95.3	95.4	0.750	0.3903	0.3104	0.4121	0.4125
		∞	$T(C)$			22.9	-	0.3847	0.3002	0.4078	0.3954

Table 8.4: GOV2: Index tuning results for relative index quality without index compression.

Table 8.5 gives tuning results for relative index quality with compressed indexes. We can get close to the result quality for top-10 results of an unpruned index with the effectiveness-oriented techniques (we even get better quality for some scenarios), for both the test and the training topics. For top-100 results, the situation is slightly worse, there is a small gap to the quality of an unpruned index (which, however, may be tolerable). For the efficiency-oriented indexes, we achieve comparable or even better precisions than the unpruned term indexes, at a reasonable index size of less than 100GB. Figures 8.10 and 8.11 depict precision and NDCG values for efficiency-oriented and effectiveness-oriented index quality goals on all (l, m) combinations from Table 8.5,

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	overlap on train	P@k on train	P@k on test	NDCG@k on train	NDCG@k on test
effective-ness oriented index quality	10	50GB	7010	0.55	49.9	49.7	0.776	0.584	0.588	0.4587	0.5130
		70GB	19810	0.30	64.9	64.7	0.854	0.596	0.576	0.4767	0.5140
		100GB	19810	0.20	98.9	98.4	0.882	0.606	0.580	0.4786	0.5139
		200GB	19810	0.05	174.5	173.5	0.924	0.614	0.584	0.4858	0.5182
		400GB	19810	0.00	306.8	304.9	0.946	0.613	0.586	0.4854	0.5210
		∞	$I(C)$			468.9	-	0.614	0.578	0.4875	0.5158
	100	70GB	20000	0.30	64.9	64.8	0.786	0.3923	0.3252	0.4155	0.4248
		100GB	20000	0.20	99.0	98.5	0.819	0.3968	0.3250	0.4189	0.4260
		200GB	19800	0.05	174.5	173.5	0.867	0.4007	0.3266	0.4223	0.4274
		400GB	19800	0.00	306.7	304.8	0.903	0.4062	0.3282	0.4259	0.4290
		∞	$I(C)$			468.9	-	0.4108	0.3338	0.4264	0.4324
efficiency-oriented index quality	10	50GB	1910	0.30	48.6	48.4	0.750	0.572	0.556	0.4549	0.5030
		70GB	1910	0.30	48.6	48.4	0.750	0.572	0.556	0.4549	0.5030
		100GB	1010	0.10	91.3	91.1	0.755	0.585	0.568	0.4614	0.5064
		200GB	510	0.00	176.1	176.0	0.752	0.614	0.556	0.4846	0.4957
		400GB	510	0.00	176.1	176.0	0.752	0.614	0.556	0.4846	0.4957
		∞	$T(C)$			14.5	-	0.585	0.538	0.4701	0.4852
	100	70GB	8900	0.30	59.6	59.4	0.750	0.3903	0.3102	0.4123	0.4124
		100GB	4100	0.15	86.1	85.7	0.751	0.3874	0.2978	0.4119	0.4001
		200GB	2100	0.05	130.1	129.7	0.752	0.3844	0.2940	0.4115	0.3976
		400GB	1200	0.00	203.5	203.2	0.750	0.3893	0.2900	0.4127	0.3948
		∞	$T(C)$			14.5	-	0.3847	0.3002	0.4078	0.3954

Table 8.5: GOV2: index tuning results for relative index quality with index compression.

for varying numbers of retrieved results. It is evident that the relative index quality approach ensures retrieval quality on test topics even without relevance assessments.

8.5.3 Query Processing with GOV2

We compared the query processing performance using pruned indexes as an input for our merge-based technique from Section 7.3.5 with the recently proposed Block-Max WAND (BMW) algorithm [DS11] as a state-of-the-art document-at-a-time algorithm. BMW requires index lists sorted in document order where entries are grouped in blocks of fixed size; for each block, the maximal score, the maximal document ID, and its size are maintained to enable skipping complete blocks during execution. We extended BMW to support proximity scores, providing two kinds of index lists as input:

- **term index lists** as described in Section 7.3.1, but ordered by document ID, and
- **proximity index lists** as described in Section 7.3.1, but ordered by document ID.

Both kinds of index lists are augmented by the block structure. We compress the document IDs by delta- and v-byte encoding and store all scores using v-byte encoding. The block size is 64 documents as used in [DS11]. We denote the respective index consisting of all term index lists for collection C by $T(C)_{BMW}$ and the index consisting of all term and proximity index lists for C by $I'(C)_{BMW}$. In our implementation of BMW, skipped blocks are not read from disk if the index list is not in memory. Note

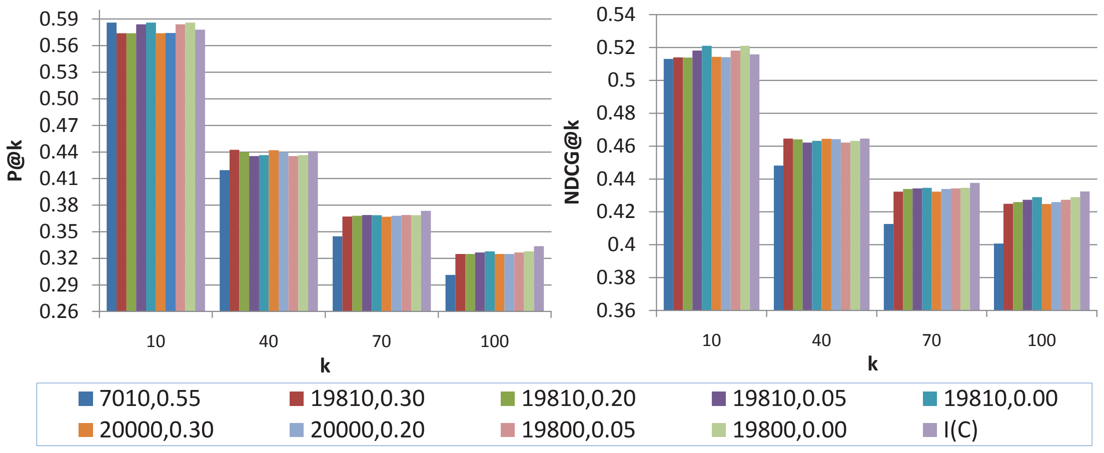


Figure 8.10: P@k and NDCG@k on test topics for effectiveness-oriented relative index quality with index compression.

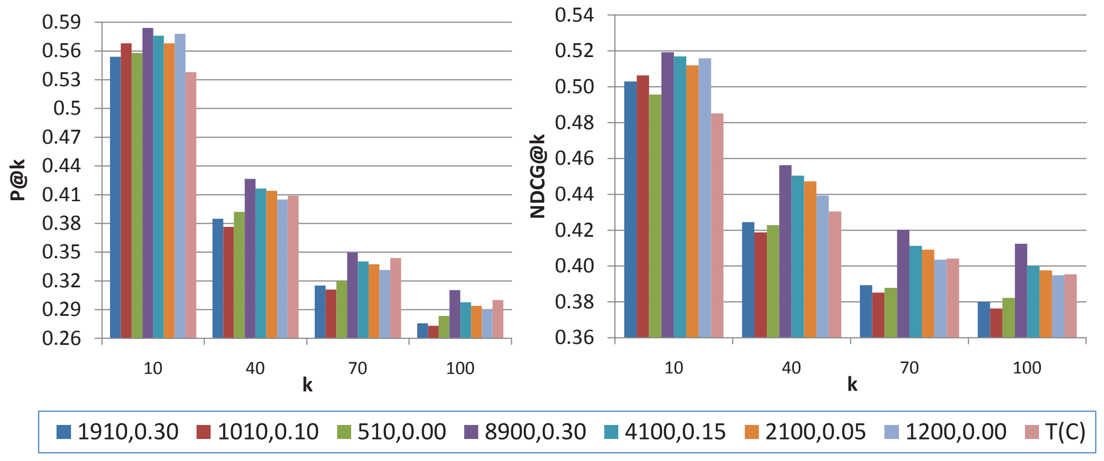


Figure 8.11: P@k and NDCG@k on test topics for efficiency-oriented relative index quality with index compression.

that it may seem appealing to simply store term position information in the term list entries and use this for proximity scoring. However, it would no longer be possible to skip blocks with this simple solution since there are no maximal proximity scores. On the other hand, using precomputed proximity lists could help to improve performance (as shown, for example, in Section 7.3 for NRA, a standard top-k algorithm and TopX in `RR-LAST` mode).

Please note that BMW evaluates queries in a disjunctive manner like the n-way merge joins described in Section 7.3.5: this means that matched documents neither have to contain all query terms nor all query terms have to appear within a maximum distance to each other. Further note that any processing algorithm would show similar performance when run on the pruned lists; the goal of this section is to compare query performance with pruned indexes to dynamic pruning on unpruned indexes.

To assess processing performance, we mainly use query processing times, but we also consider abstract cost measures such as the number of opened lists, the average number of entries or bytes read from disk. These abstract measures are not influenced by transient effects like caching or other processes running on the same machine and mask out the quality of the actual implementation. We consider two extreme settings: (1) With *hot caches*, all index lists are loaded into memory before running the first query, corresponding to the setting used in the BMW paper [DS11]. (2) With *cold caches*, all index lists are completely loaded from disk, which is ensured by flushing the file system cache before running each query, which corresponds to a very conservative setting. We will examine other caching scenarios later in this section. Processing times are measured with a single-threaded, Java-based implementation running on a single core of a single cluster node. These measurements were taken by running the complete batch of queries five times and taking the average. In addition, we invoke the garbage collector before running each query to avoid side effects caused by garbage collection during query execution. Whenever we use the average symbol $\varnothing$, we build the average over all topics of the query load under consideration. For the mere sake of completeness, we additionally provide NRA-based query performance values for runs that employ indexes without compression. In the tables, those runs are denoted $T(C)$ and $I(C)$, respectively.

For all measurements in this section, processing the training topics accessed 300 term lists and 334 combined lists, and processing the test topics accessed 144 term lists and 153 combined lists (for both pruned and non-pruned index lists).

Results with uncompressed indexes are depicted in Tables 8.6 and 8.7 for training and test topics that show the number of read index entries as well as the number of bytes and runtimes with cold and warm caches, averaged over all topics. Results with the top-k algorithm NRA on the unpruned indexes are included in the rows for $I(C)$ and $T(C)$, respectively. For the efficiency-oriented indexes, these results clearly demonstrate that query processing on the pruned indexes is up to two orders of magnitude more efficient than on the unpruned indexes. For top-10 results, we require less than 1,800 reads per topic on average with an index of 128GB size, which is less than one disk block per index list. For the effectiveness-oriented indexes, the pruned index requires

Opt.		size			size[GB]		∅reads·10^5		∅bytes·10^5		∅t_{hot}[ms]		∅t_{cold}[ms]	
goal	k	limit	$\bar{l}$	$\overline{m}$	est.	real	train	test	train	test	train	test	train	test
effective-ness oriented index quality	10	100GB	19010	0.40	96.4	96.9	0.63	0.61	6.01	5.81	28.39	31.61	226.89	243.57
		200GB	19010	0.15	170.5	170.8	0.66	0.64	6.56	6.24	28.21	27.78	232.91	245.80
		400GB	4310	0.00	396.4	396.4	0.21	0.19	2.32	2.07	9.53	8.61	177.43	182.61
		∞	$I(C)$			757.0	8.43	3.37	71.73	29.70	898.29	429.83	1368.07	1020.75
	100	100GB	10200	0.35	97.4	97.6	0.38	0.36	3.76	3.53	16.43	15.43	199.59	203.76
		200GB	17900	0.15	169.1	169.4	0.63	0.61	6.27	5.96	27.52	25.96	236.69	243.03
		400GB	4200	0.00	394.3	394.3	0.20	0.19	2.27	2.03	9.34	8.47	174.68	175.37
		∞	$I(C)$			757.0	16.81	12.91	71.73	29.70	1978.76	1628.06	2276.06	2068.93
efficiency-oriented index quality	10	100GB	5010	0.30	87.2	87.0	0.21	0.19	2.14	2.00	8.87	8.44	172.76	181.61
		200GB	310	0.05	128.1	127.9	0.02	0.02	0.21	0.20	1.19	1.11	135.03	131.86
		∞	$T(C)$			22.9	14.05	9.45	112.40	75.60	1550.40	926.13	1764.85	1311.10
	100	400GB	800	0.00	270.6	270.6	0.04	0.04	0.52	0.49	2.30	2.22	145.27	150.60
		∞	$T(C)$			22.9	20.33	15.09	112.40	75.60	3453.74	2159.41	4078.60	2669.32

Table 8.6: GOV2: query performance for absolute index quality without index compression.

Opt.		size			size[GB]		∅reads·10^5		∅bytes·10^5		∅t_{hot}[ms]		∅t_{cold}[ms]	
goal	k	limit	$\bar{l}$	$\overline{m}$	est.	real	train	test	train	test	train	test	train	test
effective-ness oriented index quality	10	100GB	12010	0.30	99.7	100.1	0.44	0.41	4.31	4.06	20.46	22.00	203.65	214.74
		200GB	19810	0.15	171.4	171.8	0.69	0.66	6.77	6.44	29.81	29.77	232.32	243.00
		400GB	19810	0.05	293.3	293.4	0.72	0.68	7.34	6.86	31.95	29.81	240.28	257.10
		∞	$I(C)$			757.0	8.43	3.37	71.73	29.70	898.29	429.83	1368.07	1020.75
	100	100GB	18100	0.35	100.0	100.4	0.61	0.59	5.83	5.61	26.35	24.85	219.07	232.50
		200GB	19800	0.15	171.4	171.8	0.69	0.66	6.76	6.44	30.07	27.99	245.75	259.52
		400GB	19800	0.05	293.3	293.4	0.72	0.68	7.34	6.86	32.07	29.47	247.53	258.77
		∞	$I(C)$			757.0	16.81	12.91	71.73	29.70	1978.76	1628.06	2276.06	2068.93
efficiency-oriented index quality	10	100GB	1910	0.30	73.1	72.7	0.09	0.08	0.96	0.87	4.05	3.77	146.11	151.57
		∞	$T(C)$			22.9	14.05	9.45	112.40	75.60	1550.40	926.13	1764.85	1311.10
	100	100GB	8800	0.30	95.3	95.4	0.33	0.32	3.37	3.15	14.67	13.53	187.33	191.77
		∞	$T(C)$			22.9	20.33	15.09	112.40	75.60	3453.74	2159.41	4078.60	2669.32

Table 8.7: GOV2: query performance for relative index quality without index compression.

up to one order of magnitude less reads than the unpruned index. For absolute index quality tuning, query performance for larger indexes is actually better, because the smaller indexes need to use long list length cutoffs, but high minscore cutoffs to meet the index size constraint, which makes query processing expensive. For relative index quality tuning, query performance for larger indexes slightly deteriorates, because the larger indexes use longer list length cutoffs but also provide higher precision values. The runtimes reported in these tables demonstrate that the theoretical cost advantage of our approach is very beneficial in practice for hot cache as well as cold cache scenarios, with average hot cache times of about 1ms for top-10 retrieval with the best efficiency-oriented index. This corresponds to two to three orders of magnitude performance advantage over standard top-k algorithm evaluation on unpruned term index lists. Unlike that, the number of read items and the runtime of our technique does not increase when retrieving more than 10 results by the nature of the merge join.

Opt.		size			size[GB]		$\varnothing$reads$\cdot 10^5$		$\varnothing$bytes$\cdot 10^5$		$\varnothing t_{hot}$[ms]		$\varnothing t_{cold}$[ms]	
goal	k	limit	$\bar{l}$	$\overline{m}$	est.	real	train	test	train	test	train	test	train	test
effective-ness oriented index quality	10	50GB	6810	0.75	48.3	48.1	0.26	0.25	1.33	1.26	8.09	8.21	90.47	85.26
		70GB	19010	0.30	64.6	64.5	0.64	0.61	2.93	2.82	17.70	17.98	136.49	133.44
		100GB	19010	0.20	98.5	98.0	0.66	0.63	3.11	2.96	17.58	18.52	138.57	135.51
		200GB	19010	0.05	173.8	172.8	0.70	0.66	3.37	3.16	18.54	18.16	151.66	139.99
		400GB	4310	0.00	249.7	248.8	0.21	0.19	1.18	1.06	6.31	5.90	82.11	73.19
		∞	$I'(C)_{BMW}$			221.0	3.96	2.58	19.13	14.57	66.65	49.88	594.42	506.89
	100	50GB	10400	0.85	49.9	49.7	0.38	0.36	1.83	1.72	11.43	11.43	115.27	107.04
		70GB	19800	0.30	64.9	64.7	0.66	0.63	3.01	2.91	17.66	18.43	143.13	141.09
		100GB	20000	0.20	99.0	98.5	0.69	0.66	3.22	3.08	18.28	19.77	149.89	152.08
		200GB	19700	0.05	174.4	173.4	0.72	0.68	3.46	3.25	19.36	18.74	157.14	148.71
		400GB	14400	0.00	295.1	293.5	0.59	0.54	2.98	2.71	16.44	15.58	149.70	144.95
		∞	$I'(C)_{BMW}$			221.0	7.07	4.79	30.72	22.96	118.67	78.49	676.27	553.75
efficiency-oriented index quality	10	50GB	6310	0.75	47.9	47.7	0.24	0.23	1.26	1.19	6.92	6.83	89.12	80.72
		70GB	5010	0.30	55.6	55.4	0.21	0.19	1.10	1.03	6.06	5.90	82.32	71.26
		100GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	13.82	13.25
		200GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	13.82	13.25
		400GB	310	0.05	94.9	94.9	0.02	0.02	0.12	0.11	0.81	0.88	13.82	13.25
		∞	$T(C)_{BMW}$			10.5	5.76	3.62	25.26	18.08	73.28	61.13	546.09	466.72
	100	50GB	10400	0.85	49.9	49.7	0.38	0.36	1.83	1.72	11.43	11.43	115.27	107.04
		70GB	6000	0.30	56.9	56.7	0.24	0.23	1.26	1.18	7.00	6.82	90.02	84.38
		100GB	3600	0.15	84.6	84.2	0.16	0.15	0.91	0.83	5.04	4.90	72.40	61.75
		200GB	900	0.00	193.8	193.5	0.05	0.05	0.32	0.30	1.89	1.86	16.42	15.86
		400GB	900	0.00	193.8	193.5	0.05	0.05	0.32	0.30	1.89	1.86	16.42	15.86
		∞	$T(C)_{BMW}$			10.5	8.21	5.40	34.17	24.92	102.50	68.76	600.41	495.58

Table 8.8: GOV2: query performance for absolute index quality with index compression.

Tables 8.8 and 8.9 show the result of the performance experiments for training and test topics. They include the number of read index entries as well as the number of bytes and runtimes with cold and hot caches, averaged over all topics. Results with BMW on unpruned indexes are included in the rows for $I'(C)_{BMW}$ and $T(C)_{BMW}$; here, we only count the number of index entries in blocks we load into main memory, not in all blocks of the lists. Due to anomalies in the corresponding tables in [BS12], we have measured the average cold cache times again and have updated Tables 8.8 and 8.9 accordingly. For the efficiency-oriented indexes, these results clearly demonstrate that query processing on the pruned indexes can be one order of magnitude more efficient than BMW for both hot and cold caches. For top-10 results, we read less than 1,800 index entries (approximately 12KB) per topic on average with an index of 94.9GB size, whereas BMW needs to access more than 360,000 entries per topic. Similar results can be achieved for top-100 results with an index of 193.5GB size. For the effectiveness-oriented indexes, the performance gap is smaller. Note that query performance for larger indexes is sometimes better because the smaller indexes need to use long list length cutoffs, but high minscore cutoffs to meet the index size constraint, which makes query processing expensive. For relative index quality tuning, query performance for larger indexes slightly deteriorates, because the larger indexes use longer list length cutoffs but usually also provide higher precision and NDCG values. The runtimes

Opt. goal	k	size limit	$\bar{l}$	$\overline{m}$	size[GB] est.	size[GB] real	⌀reads·10^5 train	⌀reads·10^5 test	⌀bytes·10^5 train	⌀bytes·10^5 test	$⌀t_{hot}$[ms] train	$⌀t_{hot}$[ms] test	$⌀t_{cold}$[ms] train	$⌀t_{cold}$[ms] test
effectiveness oriented index quality	10	50GB	7010	0.55	49.9	49.7	0.27	0.26	1.37	1.29	12.17	12.74	96.38	89.20
		70GB	19810	0.30	64.9	64.7	0.66	0.63	3.01	2.91	30.24	28.53	143.48	135.55
		100GB	19810	0.20	98.9	98.4	0.68	0.65	3.20	3.05	31.38	26.59	150.23	138.29
		200GB	19810	0.05	174.5	173.5	0.72	0.68	3.47	3.26	32.27	30.79	157.71	137.68
		400GB	19810	0.00	306.8	304.9	0.76	0.71	3.75	3.46	32.65	30.87	165.67	145.72
		∞	$I'(C)_{BMW}$			221.0	3.96	2.58	19.13	14.57	66.65	49.88	594.92	506.89
	100	50GB	10300	0.80	50.0	49.9	0.37	0.35	1.82	1.71	16.62	14.92	113.14	107.20
		70GB	20000	0.30	64.9	64.8	0.66	0.64	3.03	2.93	34.14	26.08	145.09	136.99
		100GB	20000	0.20	99.0	98.5	0.69	0.66	3.22	3.07	28.74	29.66	149.69	142.79
		200GB	19800	0.05	174.5	173.5	0.72	0.68	3.47	3.26	34.67	35.78	155.30	145.58
		400GB	19800	0.00	306.7	304.8	0.76	0.71	3.75	3.46	33.23	29.95	168.20	150.73
		∞	$I'(C)_{BMW}$			221.0	7.07	4.79	30.72	22.96	118.67	78.49	676.27	553.75
efficiency-oriented index quality	10	50GB	1910	0.30	48.6	48.4	0.09	0.08	0.52	0.47	3.79	3.03	40.36	36.04
		70GB	1910	0.30	48.6	48.4	0.09	0.08	0.52	0.47	3.79	3.03	40.36	36.04
		100GB	1010	0.10	91.3	91.1	0.05	0.05	0.33	0.31	2.23	1.95	27.93	27.38
		200GB	510	0.00	176.1	176.0	0.03	0.03	0.19	0.18	1.72	1.13	19.62	18.27
		400GB	510	0.00	176.1	176.0	0.03	0.03	0.19	0.18	1.72	1.13	19.62	18.27
		∞	$T(C)_{BMW}$			10.5	5.76	3.62	25.26	18.08	73.28	61.13	546.09	466.72
	100	50GB	10300	0.80	50.0	49.9	0.37	0.35	1.82	1.71	16.62	14.92	113.14	107.20
		70GB	8900	0.30	59.6	59.4	0.34	0.32	1.69	1.59	15.17	13.72	115.14	104.44
		100GB	4100	0.15	86.1	85.7	0.18	0.17	1.01	0.92	7.40	8.21	78.90	70.51
		200GB	2100	0.05	130.1	129.7	0.10	0.10	0.62	0.57	5.00	5.48	53.33	47.13
		400GB	1200	0.00	203.5	203.2	0.06	0.06	0.41	0.38	2.82	3.24	36.07	27.97
		∞	$T(C)_{BMW}$			10.5	8.21	5.40	34.17	24.92	102.50	68.76	600.41	495.58

Table 8.9: GOV2: query performance for relative index quality with index compression.

reported in these tables demonstrate that the theoretical cost advantage of our approach is very beneficial in practice for hot cache as well as cold cache scenarios, with average hot cache times of about 1ms for top-10 retrieval with the best efficiency-oriented index. Please note that we have used pruned test indexes in the tables. The (310,0.05) test index needs on average 13.82ms and 13.25ms in a cold cache scenario for training and test topics, respectively. This seems to be extremely fast and is probably influenced by the dense arrangement of the test index structures on the hard disk and the resulting non-controllable disk caching side effects. To allow a comparison, we have built the (310,0.05) full index: for training and test topics $⌀t_{cold}$ values are 75.13ms and 68.65ms, respectively. Unlike BMW, the number of read entries and the runtime of our technique does not increase when retrieving more than 10 results by the nature of the merge join (however at the price of a slightly reduced result quality).

As an interesting side result, we see that the additional proximity lists can sometimes improve query performance for BMW because they allow tighter score bounds, which is similar to the earlier results in Chapter 7 for the standard top-*k* algorithm NRA and TopX in `RR-LAST` mode. For top-10 retrieval, BMW with term and proximity lists (denoted as $I'(C)_{BMW}$ in Table 8.8) takes on average 66ms with hot caches for the training topics and reads on average 396K entries, whereas using only term lists ($T(C)_{BMW}$) takes on average 73ms and reads on average 576K entries. With cold caches, BMW with only term indexes is better due to the expensiveness of opening more index lists (546ms vs 594ms).

$\bar{l}$	$\overline{m}$	cache size[MB]	cache hit ratio [bytes]	cache hit ratio [#lists]	#non-cached lists	$\varnothing t_{warm}$[ms]
310	0.05	8	28.98%	29.29%	161,393	39.85
310	0.05	16	37.05%	37.08%	143,613	36.04
310	0.05	32	44.36%	43.89%	128,069	32.70
310	0.05	64	50.54%	49.39%	115,525	29.67
310	0.05	1024	54.44%	52.77%	107,801	28.92

Table 8.10: Efficiency Track: real system performance, merge join, various LRU cache sizes with a (310,0.05) full index.

For efficiency reasons, storing position information in the term-only index to compute proximity scores on the fly as a document is encountered is not an option for us. As argued in Section 7.2.2, that approach is not feasible for top-k style processing as it is not possible to compute tight score bounds for candidates which in turn disables dynamic pruning.

To compute the top-k results efficiently, we need to precompute proximity information into index lists that can be sequentially scanned and compute tight score bounds for early pruning. As briefly discussed in Section 6.3, an alternative would be to first determine a set of candidate documents with 'good' term list scores and later re-rank only the candidate documents by computing proximity scores from their position information (cf. Section 2.4.1 for an example). This requires a large enough set of candidate documents in the first step, which is potentially expensive to compute - if we choose the set too small, we may leave out potentially relevant documents and decrease result quality for the top-k results. As shown in Table 8.8 for BMW ($T(C)_{BMW}$), this can easily cause high additional processing costs.

In a second line of experiments, we ran the 50,000 queries from the TREC Terabyte Efficiency Track 2005 with the fastest index configuration determined by efficiency-oriented index-tuning, the (310, 0.05) full index setting, comparing it again to BMW. In addition to the hot and cold cache settings used before, we also consider *warm caches*, a more realistic simulation of a running system. We implemented an LRU cache of configurable size to store the least recently used index lists. This LRU cache was emptied before running the first query; we then ran all queries sequentially. To minimize side effects caused by file system caching and garbage collector activities during query processing, we emptied the file system cache and invoked the garbage collector before executing each query. In this scenario that corresponds to a steady-state execution in a running search engine, processing with the (310, 0.05) full index takes less than 30ms on average for an LRU cache size of 64MB, compared to an average of 0.7ms for hot caches and 127ms for cold caches, respectively.

Table 8.10 shows performance values for query processing with a (310, 0.05) full index for different cache sizes. It depicts the cache hit ratio both for the number of read bytes (cache hit ratio[bytes]) and number of read lists (cache hit ratio[#lists]) as well as the number of non-cached lists and the average warm cache running times. Even for a very small cache size of 8MB, we need less than 40ms on average to process

		cache	cache hit ratio		#non-cached		
k	index	size[MB]	[bytes]	[#lists]	lists	#read blocks	$\varnothing t_{warm}$[ms]
10	$T(C)_{BMW}$	64	3.62%	2.15%	115,938	397,735,689	204.63
100	$T(C)_{BMW}$	64	3.62%	2.15%	115,938	573,001,133	259.01
10	$T(C)_{BMW}$	1024	46.39%	26.58%	86,990	397,735,689	173.16
10	$I'(C)_{BMW}$	1024	44.02%	14.57%	197,638	312,500,850	206.36

Table 8.11: Efficiency Track: real system performance, BMW, various LRU cache sizes.

a query, and starting with a cache size of 64MB, we need less than 30ms on average. Note that the overall number of index lists used in this experiment is 228,247.

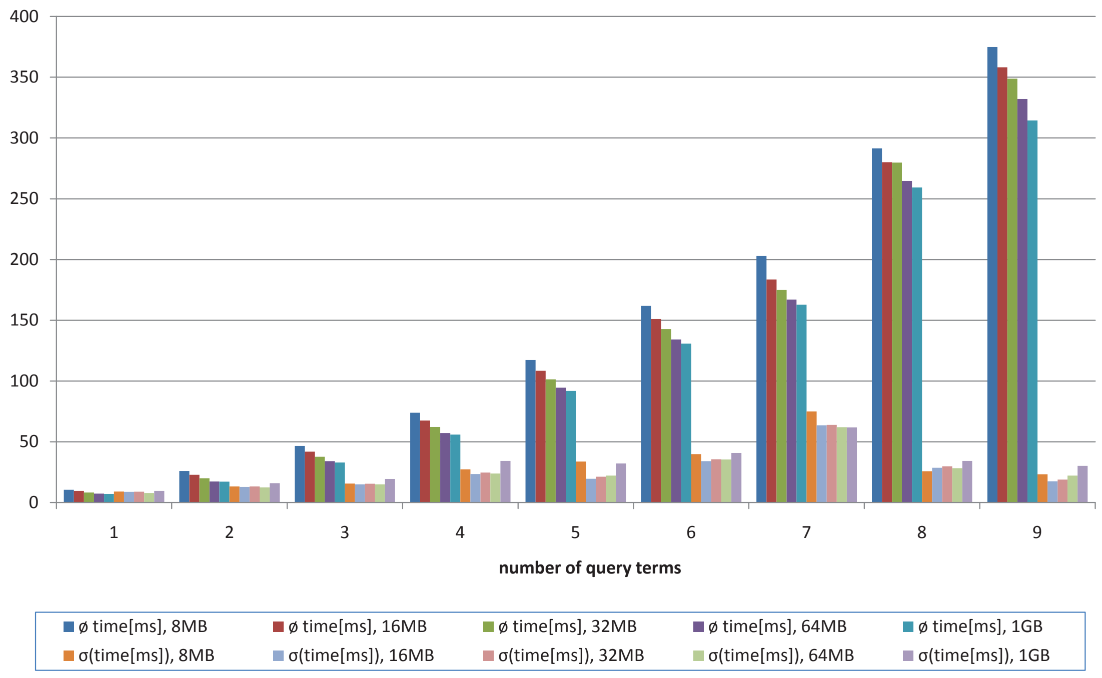


Figure 8.12: Efficiency Track: real system performance for a (310,0.05) full index for various query and LRU cache sizes.

Figure 8.12 depicts the average running times and their standard deviations depending on the number of keywords in the query and the cache size. The larger the cache size, the higher the cache hit ratio which lowers the average processing time. As expected, the average running time is monotonous in the number of query terms as more query terms potentially lead to more fetched index lists at processing time. However the standard deviation of running times for a given query length is low (and does not depend on the cache size) such that the average running time is usually a good approximation for the expected running time of a query.

Table 8.11 shows performance values for query processing with BMW on $T(C)_{BMW}$ and $I'(C)_{BMW}$ indexes with varying cache sizes. The number of processed lists for this query load amounts to 118,483 for $T(C)_{BMW}$ which consist of 1,061,659,742 blocks in

196.9GB size, and 231,346 for $I'(C)_{BMW}$ indexes which consist of 1,099,312,741 blocks in 202.5GB size[9]. As expected, increased cache sizes help to speed up query processing (204.63ms vs 173.16ms for 64MB vs 1GB LRU cache size with $T(C)_{BMW}$). $I'(C)_{BMW}$ is slower than $T(C)_{BMW}$ at the same cache size: although the number of read blocks decreases, more non-cached lists have to be loaded. We see that the run time increases with growing k. While processing queries with $T(C)_{BMW}$ with 64MB LRU cache size takes 204.63ms to retrieve k=10 results, the same index requires 259.01ms to retrieve k=100 results. In contrast, the run time is independent of the result set cardinality for our pruned indexes as they are processed completely by an n-way merge join.

If we compare the warm cache performance of our pruned lists processed in a merge join algorithm (Table 8.10) to that of BMW with a $T(C)_{BMW}$ index (Table 8.11) at an LRU cache size of 64MB, we observe that the number of non-cached lists that have to be fetched from hard disk is similar (115,525 vs 115,938). Anyway we achieve a speedup of a factor of 7 ($\varnothing t_{warm}$=204.63ms) for top-10 and a factor of 9 for top-100 retrieval ($\varnothing t_{warm}$=259.63ms), since the cache hit ratio measured in bytes is about 50% for our approach compared to less than 4% for the BMW approach. To achieve a similar cache hit ratio for the $T(C)_{BMW}$ index as for our approach, we need to increase the LRU cache size to 1GB. This means that, compared to our approach, we need 16 times as much cache at a processing speed that is 5 times slower.

8.5.4 Log-based Pruning with GOV2

We evaluated our log-based technique for pruning term pairs from the index, using the same training and test queries as before and the AOL query log. Note that all indexes in this section are uncompressed as we consider log-based pruning and compression of indexes as orthogonal ways to shrink index sizes whose effects are shown separately. Table 8.12 shows index tuning results for t=1, i.e., materializing combined lists for

Opt.		size			size[GB]		P@k on	
goal	k	limit	$\bar{l}$	$\overline{m}$	est.	real	train	test
effectiveness-oriented index quality	10	100GB	5010	0.00	96.0	96.0	0.610	0.586
	100	100GB	19800	0.10	93.6	93.6	0.3927	0.3192
		400GB	20000	0.00	293.4	293.4	0.3969	0.3196
efficiency-oriented index quality	10	50GB	2210	0.15	47.8	47.8	0.538	0.462
		100GB	1810	0.00	64.7	64.7	0.587	0.554
	100	50GB	7200	0.35	50.0	50.2	0.3771	0.3036
		100GB	3800	0.00	86.2	86.2	0.3847	0.2980

Table 8.12: Index tuning results with log-based pruning (t=1) for absolute index quality.

term pairs that occur at least once in the query log. It is evident that using log-based pruning helps to get smaller index sizes for the efficiency-based techniques. The index size reduces down to 47.8GB, which is approximately twice the size of the unpruned, uncompressed term index (22.9GB). The result quality of indexes created by log-based

[9]There are more lists here because our pruning technique may completely drop a pair when all entries in its list have a score below the minscore threshold.

pruning remains similar as for the unpruned, uncompressed term index. However, index tuning using log-based pruning results in much longer index lists than index tuning without log-based pruning. The longer lists in turn affect runtime (cf. Table 8.13):

Opt. goal	k	$\bar{l}$	$\overline{m}$	P@k train	P@k test	⌀reads·10^5 train	⌀reads·10^5 test	⌀t_{hot}[ms] train	⌀t_{hot}[ms] test	⌀t_{cold}[ms] train	⌀t_{cold}[ms] test
efficiency- oriented	10	2210	0.15	0.538	0.462	1.22	1.97	162.4	241.6	427.1	514.2
		1810	0.00	0.587	0.554	1.20	1.96	162.2	229.6	441.3	525.7
index quality	100	7200	0.35	0.3771	0.3036	1.38	2.12	229.4	388.8	504.3	685.7
		3800	0.00	0.3847	0.2980	1.29	2.03	255.6	341.8	471.0	649.7

Table 8.13: Query performance with log-based pruning (t=1) for absolute index quality.

compared to the best results without log-based pruning, query processing takes an order of magnitude longer. The longer run times are due to the score-ordered part of the unpruned term lists as detailed in Section 8.4 which may be processed to a large extent in the NRA phase to preserve retrieval quality if all corresponding combined lists are missing. Anyway, it is still faster than BMW with unpruned $T(C)$ in the cold cache setting. We could not achieve the quality goal for the effectiveness-based methods as there were not enough combined lists left to boost quality enough; we did not evaluate performance for these settings. Log-based pruning therefore mostly serves to reduce index size in situations with strong resource constraints where indexes need to be loaded from disk. Here, it can still improve execution cost, while result quality stays comparable to a term-only index.

8.5.5 Summary of Conclusions and Limitations of the Approach

Finding ($\bar{l}$,$\overline{m}$) parameters to build pruned GOV2 indexes is reasonably fast and takes about five hours, while building a final full index is dependent on the resulting parameters and takes less than five hours for ($\bar{l}$,$\overline{m}$)=(310,0.05). Already very short list prefixes of term and combined lists are sufficient to yield a result quality comparable to the one of unpruned term lists. This comes at the expense of more opened lists but saves on the number of read bytes and tuples, respectively. As shown for BMW, processing unpruned term indexes using dynamic pruning techniques is more expensive than processing pruned term and combined lists in an n-way merge join. P@k' and NDCG@k' values of pruned indexes tuned for k results do not degrade much for larger result set cardinalities k'. We show that, in the absence of relevance assessments, we can use the overlap between top-k results on pruned term and combined lists and the top-k results on unpruned lists as a substitute for the relevance assessments. The relative index quality approach ensures retrieval quality on test topics even without relevance assessments for the training topics – it works better for early P@k and NDCG@k, but does not degrade much for larger k. Processing performance for training and test topics is excellent compared to the BMW algorithm.

For a bigger query load, the 50,000 TREC Terabyte Efficiency Track queries from 2005, our experiments show the viability of our approach at an LRU cache size of 64MB: although the number of non-cached lists for our pruned index and the $T(C)_{BMW}$ index is comparable, our processing speed is 7 and 9 times faster for top-10 and top-100 retrieval, respectively, as the longer lists of the $T(C)_{BMW}$ index require more cache space. To achieve a similar cache ratio for the $T(C)_{BMW}$ index as for our pruned index, we would need to increase the LRU cache size to 1GB, at a processing speed which is still 5 times slower than for our approach.

Log-based pruning helps to get smaller resulting indexes at similar result quality at the expense of increased list lengths and an increased number of loaded lists which incurs increased running time during query processing.

8.5.6 Results with ClueWeb09

			size[GB]		P@k		$\varnothing$reads$\cdot 10^5$		$\varnothing$bytes$\cdot 10^5$		$\varnothing t_{hot}[ms]$		$\varnothing t_{cold}[ms]$	
Opt. goal	k	$(\bar{l}, \overline{m})$	est.	real	train	test	train	test	train	test	train	test	train	test
effect.-o.	10	(1410,1.0)	640.2	636.2	0.200	-	0.04	0.04	0.24	0.18	2.32	1.25	67.35	56.38
index qual.	100	(4600,0.3)	979.0	977.9	0.1344	-	0.14	0.11	0.75	0.56	5.35	3.33	83.69	61.44
effic.-o.	10	(410,1.0)	503.6	499.8	0.190	0.2292	0.01	0.01	0.07	0.06	0.77	0.47	79.61	71.15
index qual.	100	(400,1.0)	464.6	462.8	0.1170	-	0.01	0.01	0.07	0.06	0.63	0.44	69.05	55.60

Table 8.14: ClueWeb09: index tuning results for absolute index quality and evaluation of query performance, size limit set to S=1TB.

To demonstrate the scalability of our index tuning approach, we carried out experiments on the ClueWeb09 collection similar to the ones shown in Section 8.5.2 for GOV2. For all experiments we keep the index size limit of S=1TB fixed which corresponds to about 17% of the size of the uncompressed spam-reduced English part of the ClueWeb09 collection (cf. Section 3.2.1 for details). We have published the corresponding results in [BS10].

Again, we consider two baselines, the unpruned $T(C)$ and $I(C)$ indexes; however, as building the unpruned $I(C)$ index exceeded our disk capacity, we had to limit each index list to the first 20 million entries. We expect that this restriction will—if at all—have only a negligibly small influence on the result quality. The result quality of results created with these baselines was assessed using the available assessments for the training topics; here, P@10 was 0.180 for $T(C)$ and 0.198 for $I(C)$, and P@100 was 0.1110 for $T(C)$ and 0.1324 for $I(C)$. For the test topics, we submitted runs created for the baseline indexes to the 2010 TREC Web Track, Ad Hoc Task, which yielded a P@10 of 0.2250 for both $T(C)$ and $I(C)$ indexes, which was somehow unexpected (we had expected $I(C)$ to yield a higher precision than $T(C)$). This may be partially caused by sparser relevance assessments for the Web Track Topics, partially by giving a lower assessment priority to our two baseline runs compared to our third run (which was created with one of the pruned indexes). For P@100 our expectations are met: $I(C)$ yields a precision of 0.1358 compared to $T(C)$ which yields a precision of 0.1294.

Table 8.14 shows the results for absolute index quality tuning on the training topics for compressed ClueWeb09 indexes. With index parameters tuned for efficiency and top-10 document retrieval, the best index configuration turns out to be (410, 1.00) at an index size of less than 500GB. Processing training topics with this pruned index requires 1,302 reads on average per topic and takes less than 1ms for hot and 80ms for cold caches, providing a result quality comparable to $T(C)$. Processing test topics with the same index is even slightly faster due to shorter index lists (1,045 reads on average), reflected in improved hot cache times and cold cache times of about 70ms. The corresponding run has been submitted to the 2010 TREC Web Track, Ad Hoc Task as well, with a P@10 of 0.2292, which is slightly higher than the P@10 of $T(C)$ and slightly higher than the P@10 value for the training topics. As NDCG@k has not been used as retrieval quality metric in the 2010 TREC Web Track, Ad Hoc Task, we only report precision values.

Efficiency-oriented indexes for top-100 document retrieval require less than 500GB disk space and provide running times of less than 1ms for hot caches and around 70ms for cold caches on the training topics at a result quality comparable to the $T(C)$ baseline run. Query processing on the test topics is again faster, with 55ms on average with cold caches.

Our effectiveness-oriented index tuned for top-10 retrieval (1410, 1.00) requires 640GB and provides a retrieval quality comparable to $I(C)$. Query execution takes about 2ms for the training topics and slightly more than 1ms for the test topics for hot caches. Effectiveness-oriented indexes for top-100 retrieval require less than 1TB disk space and thus stay within our index size limit, providing again a result quality comparable to $I(C)$. Here, query execution takes about 5ms and 3ms for hot caches, whereas cold cache times range below 85ms and 65ms for training and test topics, respectively. The results show that the size estimator also works effectively on the ClueWeb09 collection with only minor overestimation.

			size[GB]				⌀reads	⌀bytes	$\varnothing t_{hot}$	$\varnothing t_{cold}$
Opt. goal	k	$(\bar{l}, \overline{m})$	est.	real	overlap	P@k	$\cdot 10^4$	$\cdot 10^5$	[ms]	[ms]
effect.-o.	10	(9810,1.00)	927.9	923.6	0.986	0.198	2.89	1.45	8.46	152.17
index qual.	100	(19000,0.80)	1008.3	1006.5	0.972	0.1330	5.52	2.72	13.77	135.93
effic.-o.	10	(110,1.00)	395.3	391.6	0.856	0.160	0.04	0.02	0.28	70.89
index qual.	100	(200,1.00)	408.1	407.6	0.781	0.1010	0.06	0.04	0.43	82.95

Table 8.15: ClueWeb09: index tuning results for relative index quality and evaluation of query performance, size limit set to S=1TB.

Table 8.15 shows the results for relative index quality tuning on ClueWeb09 with the training topics. While the effectiveness-oriented approaches result in indexes which deliver result quality comparable to $I(C)$ (at the price of longer lists compared to absolute index quality), result quality with the efficiency-oriented indexes falls shortly behind BM25 score quality, but the difference would still be tolerable in applications. We assume that this effect can at least partly be attributed to the fact that relevance assessments from TREC 2009 are very sparse compared to those from earlier years; unassessed documents contribute to the overlap with the groundtruth, but do not increase precision values if they are in the result list of a query, even though a user may consider them relevant.

Although the indexed part of the ClueWeb09 collection is one order of magnitude larger in size than GOV2 (6TB vs. 426GB uncompressed), the required index space does not grow as fast as the collection (e.g., index size grows from 94.9GB to 499.8GB for the efficiency setting (310, 0.05) on GOV2 compared to (410, 1.00) on ClueWeb09). For absolute index quality tuning, the indexes tend to have shorter list lengths on ClueWeb09 such that query processing is often even faster on ClueWeb09 indexes.

8.5.7 Results with INEX 2009

This part is based on our participation in INEX 2009 [BS09] which describes our efforts that apply our tuning framework to the INEX 2009 test bed for the Efficiency Track. We tune the index structures for different choices of result size k. To allow comparison as to retrieval quality with non-pruned index structures, we also depict our results from the Ad Hoc Track. The scoring model we used in INEX 2009 corresponds to the one we used in INEX 2008 [BST08], this time retrieving article elements only. Details about the scoring model can be found in Section 5.2.4.

Ad Hoc Track

For our contribution to the Ad Hoc Track, we removed all tags from the XML documents in the official INEX 2009 collection and worked on their textual content only. The last two runs have been submitted to INEX 2009, the first is the non-submitted baseline:

- `MPII-COArBM'`: a content-only (CO) run that considers the stemmed terms in the title of a topic (including the terms in phrases, but not their sequence) except terms in negations and stop words. We restrict the retrieval to the top-level article elements and compute the 1,500 articles with the highest $score_{\text{BM25}}$ value as described in our contribution to INEX 2008 [BST08]. Note that this approach corresponds to standard document-level retrieval. This run is the actual non-submitted baseline to enable a comparison to the submitted runs which all use proximity information. The corresponding run in Section 5.2.6 has been named `TopX-CO-Baseline-articleOnly`.
- `MPII-COArBP`: a CO run which aims to retrieve the 1,500 articles with the highest $score_{\text{BM25}}+score_{prox}$ values, where $score_{prox}$ is calculated based on all possible stemmed term pairs in the title of a topic (including the terms in phrases, but not their sequence) except terms in negations and stop words.
- `MPII-COArBPP`: a CO run which is similar to `MPII-COArBP` but calculates the $score_{prox}$ part based on a selection of stemmed term pairs. Stemmed term pairs are selected as follows: we consider all stemmed tokens in phrases that occur both in the phrasetitle and in the title and are no stop words. The modified phrases in the phrasetitle are considered one at a time to combine term pairs usable to calculate $score_{prox}$. If the phrasetitle is empty, we use approach `MPII-COArBP`.

The results in Table 8.16 show that computing our proximity score with a subset of term pairs based on information taken from the phrasetitles (`MPII-COArBPP`) does not

run	iP[0.00]	iP[0.01]	iP[0.05]	iP[0.10]	MAiP
MPII-COArBM'	0.5483	0.5398	0.5112	0.4523	0.2392
MPII-COArBP	0.5603	0.5516(26)	0.5361	0.4692	0.2575
MPII-COArBPP	0.5563	0.5477(28)	0.5283	0.4681	0.2566

Table 8.16: Results for the Ad Hoc Track: interpolated precision at different recall levels (ranks for iP[0.01] are in parentheses) and mean average interpolated precision.

improve the iP values compared to using all term pairs (`MPII-COArBP`). As expected, `MPII-COArBP` leads to a slight improvement over `MPII-COArBM'`.

Efficiency Track

In the following, we describe our effort in INEX 2009 to tune our index structures for efficient query processing, taking into account the expected retrieval quality and index size. After that we briefly explain the approaches used by the other participants in the Efficiency Track and conclude with the results of the Efficiency Track.

Like for index tuning with the GOV2 and ClueWeb09 collection, we aim to prune TLs and CLs after a fixed number of entries per list (plus an optional minimum score requirement for CLs) and employ them as input to a merge join.

To measure retrieval quality, one usually compares the retrieval results with a set of relevance assessments. As at the time of index tuning we did not have any relevance assessments and we aim at maximum query processing speed, we tuned for efficiency-oriented relative index quality. To this end, for each number of results k required by INEX ($k \in \{15,150,1500\}$), we first built up a groundtruth as a substitute for relevance assessments. That groundtruth consists of the top-k results obtained through processing the $I(C)$ index. Note that this corresponds to the k highest scoring results of `MPII-COArBP`.

We have found in Section 8.5.2 for the GOV2 collection that it was reasonable to use an overlap of α=75% between the top-k documents obtained by query processing on pruned TLs and CLs and the top-k documents of the groundtruth. This is enough to achieve the retrieval quality of $T(C)$, i.e., BM25 retrieval quality. (Note that the overlap is computed by the amount of overlapping documents and is not based on the number of characters returned.)

The optimization process follows the description given in Section 8.3.1. Please note that for our submission to INEX 2009 we used an early implementation of the tuning framework described in Section 8.3.2 which supported only uncompressed indexes: for all list lengths l ranging between 10 and 20,000 (step size of 100) and minimal score cutoffs between 0 and 1 (step size 0.05), we estimate the index size first by hashcode-based sampling 1% of all terms and term pairs. In our experiments, we restrict the processing of the query load to those indexes that meet the index size constraint set to S=100 GB.

Table 8.17 presents the tuning results based on Type A queries with efficiency-

k	$(\bar{l}, \bar{m})$	overlap	est. size[GB]	$N_t(I)$	$N_c(I)$	$s_t \cdot N_t(I)$[MB]	$s_c \cdot N_c(I)$[GB]
15	(210,0.00)	0.7629	62.8	$66.4 \cdot 10^6$	$3.55 \cdot 10^9$	507	52.9
150	(610,0.00)	0.7639	74.3	$91.3 \cdot 10^6$	$4.31 \cdot 10^9$	697	64.2
1500	(1810,0.00)	0.754	84.4	$123.3 \cdot 10^6$	$4.97 \cdot 10^9$	941	74.1

Table 8.17: Tuning results based on Type A queries with efficiency-oriented relative index tuning, uncompressed indexes.

oriented relative index tuning for the three result cardinalities k. The size of the access structure $a_t \cdot K_t(I) + a_c \cdot K_c(I)$ is estimated to 9.4 GB for all choices of k with $K_c(I) = 7.62 \cdot 10^8$ and an average term pair length of 12.17 in the sample.

run	$(\bar{l}, \bar{m})$	$\varnothing t_{hot}$[ms]	$\varnothing t_{cold}$[ms]	**iP[0.00]**	**iP[0.01]**	**iP[0.05]**	**iP[0.10]**	**MAiP**
MPII-eff-15	(210,0.00)	8.8	216.5	0.575	0.559	0.511	0.400	0.177
MPII-eff-150	(610,0.00)	13.2	242.5	0.574	0.560	0.531	0.466	0.233
MPII-eff-1500	(1810,0.00)	27.1	287.0	0.566	0.553	0.532	0.464	0.248

Table 8.18: Efficiency Track results, type A queries.

Table 8.18 shows the results of the tuned index structures for type A queries. For performance reasons, tuning was carried out using the type A queries only. To process type B queries, we used the same pruned indexes. `MPII-eff-k` depicts the optimal list lengths for different choices of k, the average cold and hot cache running times, and interpolated precision values at different recall levels. While measuring the cold cache running times, we have emptied the filesystem cache after each query execution, not just after each batch. To collect the hot cache running times, in a first round we fill the cache by processing the complete query load and measure the running times in the second round. The difference between the cold and hot cache running times can be considered as I/O time. Queries are processed using the pruned index structures which

run	$(\bar{l}, \bar{m})$	$\varnothing t_{hot}$[ms]	$\varnothing t_{cold}$[ms]	**iP[0.00]**	**iP[0.01]**	**iP[0.05]**	**iP[0.10]**	**MAiP**
MPII-eff-15	(210,0.00)	604.3	7,630.4	0.374	0.356	0.304	0.272	0.099
MPII-eff-150	(610,0.00)	922.7	10,235.3	0.391	0.379	0.338	0.315	0.157
MPII-eff-1500	(1810,0.00)	1,492.3	12,979.9	0.391	0.379	0.337	0.316	0.162

Table 8.19: Efficiency Track results, type B queries.

have been reordered by docid to allow merge join query processing. The pruned index is created by Hadoop and, in that early version, stored in a MapFile which is accessed by Hadoop in a non-optimized way during query execution: hence, there is still room for performance improvements. These performance improvements have been realized in later implementations (see experiments with other test beds) by means of our own file-based inverted list implementation and access methods detailed in Section 8.2. It turns out that already very short list prefixes are sufficient to lead to a result quality comparable to `MPII-COArBP` at early recall levels (until iP[0.01]) and to `MPII-COArBM'` at later recall levels.

Table 8.19 shows the results of the tuned index structures for type B queries. It is clear that in our setting type B queries that consist of partly more than 100 keywords cannot be executed as fast as type A queries. Many thousands of possible pruned CLs per query have to be fetched from hard disk before the evaluation can start.

Other participants In the following, we will describe the approaches pursued by the other participants in the Efficiency Track of INEX 2009.

Spirix: SPIRIX [WK09] is a P2P system which uses distributed search techniques for XML retrieval and splits collection, index, and search load over the P2P network. The employed P2P protocol is based on a Distributed Hash Table (DHT). The authors exploit XML structure to reduce the number of messages sent between peers. To compute the structural similarity with indexed articles for CAS queries, the authors use four groups of functions, which are used in different combinations for ranking and routing. The authors have implemented adaptions of several scoring models, namely the BM25, BM25E, and $tf \cdot idf$ model. SPIRIX has participated in the Ad Hoc and Efficiency track where the precision values were competitive with centralized solutions.

The authors claim that they can reduce the total amount of different structures to 1539. There is no information about the number of structures in the full index such that the extent of the reduction remains unclear. The system significantly improves on early precision measures when it uses structural similarity. However an improvement from iP[0.01]=59% to 59.9% comes at the price of about 50 times slower query processing.

In contrast to the other participants in the Efficiency Track who aim at providing fast query execution times, the authors redefine efficiency as getting the P2P system to scale, i.e., load balancing on large collections.

MPII-TopX2: MPII-TopX2 used in INEX 2009 [TAS09] is based on the earlier reimplementation of TopX for INEX 2008 [TAS08] and extends it by a new distributed XML indexing component. It supports a CAS-specific distributed index structure with a parallelization of all indexing steps. The overall time for indexing, which is done in a 3-pass process, amounts to 20 hours on a single node system, and 4 hours on a cluster with 16 nodes.

Retrieval modes include the Article mode (retrieves only article elements), CO Mode (retrieves any kind of elements), and CAS Mode (supports path queries with NEXI or XPath2.0 syntax). Entire lists or their prefixes, respectively, can be cached; the decoded and decompressed data structures can be reused by MPII-TopX2. Keys are tag-term pairs with a term propagation upwards in the XML tree. tf and ef values are computed for each tag-term pair. Due to the large collection size, collection-wide statistics are approximated. The scoring model in use is an XML-specific extension to BM25. The index uses an inverted block structure and compresses the blocks into a customized compact binary format.

The Otago System: The system developed at the University of Otago [TJG09] uses a dictionary of terms organized in two levels: the first level stores the first four bytes and the length of every term string, and the position to retrieve the term block that belongs to the term prefix. Terms with the same four bytes prefix are stored in the same term block which stores terms statistics: these include ctf and *df* values, the offset to locate the postings list, the length of the postings list, the uncompressed length of the postings list, and the position to locate the term suffix which is stored at the end of the block. At start-up, only the first level dictionary is loaded into memory. Query processing allows to set two parameters, namely lower-k and upper-K. While lower-k specifies how many documents to return, upper-K specifies how many documents to read from each tf-sorted postings list. (If there are ties, the postings with the same tf-value as the Kth posting are also evaluated.) When upper-K is specified, the complete postings list is decompressed, but only the documents with the highest tf-values are processed. This is similar to impact-layered indexes presented in Section 6.2.1. Each tf-layer stores the document ids in increasing order and compresses them by delta-encoding. Postings are compressed by v-byte encoding. The authors employ a special version of the quick sort algorithm that partitions the accumulators by their score so that only the top-partition has to be sorted.

The employed scoring model is a modified version of BM25. Only operating system caching is used with the disk cache flushed before each run. A memory layer of 5.3GB is allocated with a usage of 97%. Lower-k is chosen as 15, 150, and 1,500 as required by the Efficiency Track. Each choice of lower-k is combined with upper-K set to 1, 15, 150, 1,500, 15,000, 150,000, and 1,500,000 which generates 21 runs. The runs which yield the highest MAiP of 29% and 30%, respectively, set lower-k to 1,500 and use an upper-K of at least 15,000. Early precision (iP[0.01]) values are good unless upper-K is chosen small (i.e., too few entries per postings list have been read); peak values between 58% and 60% are achieved for upper-K choices of at least 15,000. The average run time is split into I/O and CPU part. For Type A topics, the I/O costs are more or less constant (between 56ms and 58ms on average per topic), whereas CPU costs increase with increasing Upper-K (around 20ms for Upper-K$\leq$1,500, up to about 65ms for upper-K=1,500,000). For Type B topics, the I/O costs are again very similar, as for all choices of lower-k and upper-K the same number of postings is retrieved from disk, causing the same disk I/O (between 325ms and 350ms). The CPU costs are similar for upper-K values of at most 150 (around 30ms) independent of the lower-k choice. Due to the increased number of postings lists for Type B topics, compared to Type A topics, the CPU time increases way more (up to 217ms for upper-K=1,500,000). The best MAiP value of 18% is achieved for lower-k=1,500 and upper-K=150,000. Run time is dominated by the I/O costs. Lower-k values above 15,000 lead to an increased run time to sort the top-partition of the accumulators.

Efficiency Track Results Figures 8.13 and 8.14 describe the performance of the submitted runs in terms of efficiency and effectiveness (MAiP metrics) for type A and type B topics, respectively.

Figures 8.15 and 8.16 describe the performance of the submitted runs in terms of efficiency and effectiveness (iP[0.01] metrics) for type A and type B topics, respectively.

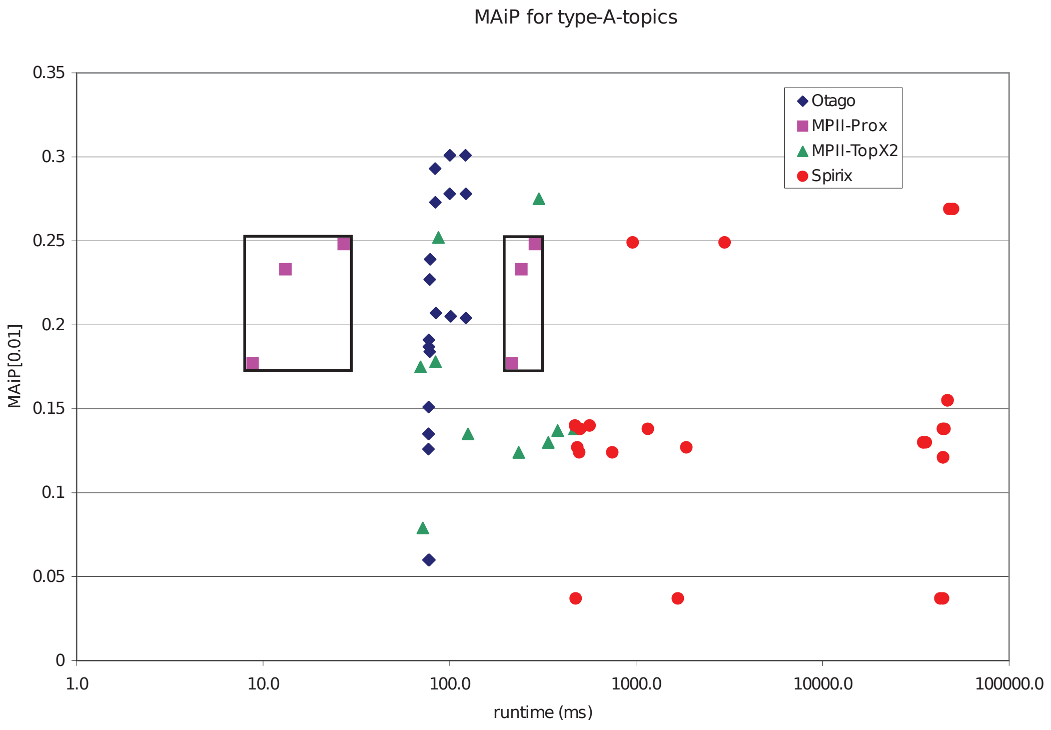


Figure 8.13: MAiP values: type A queries.

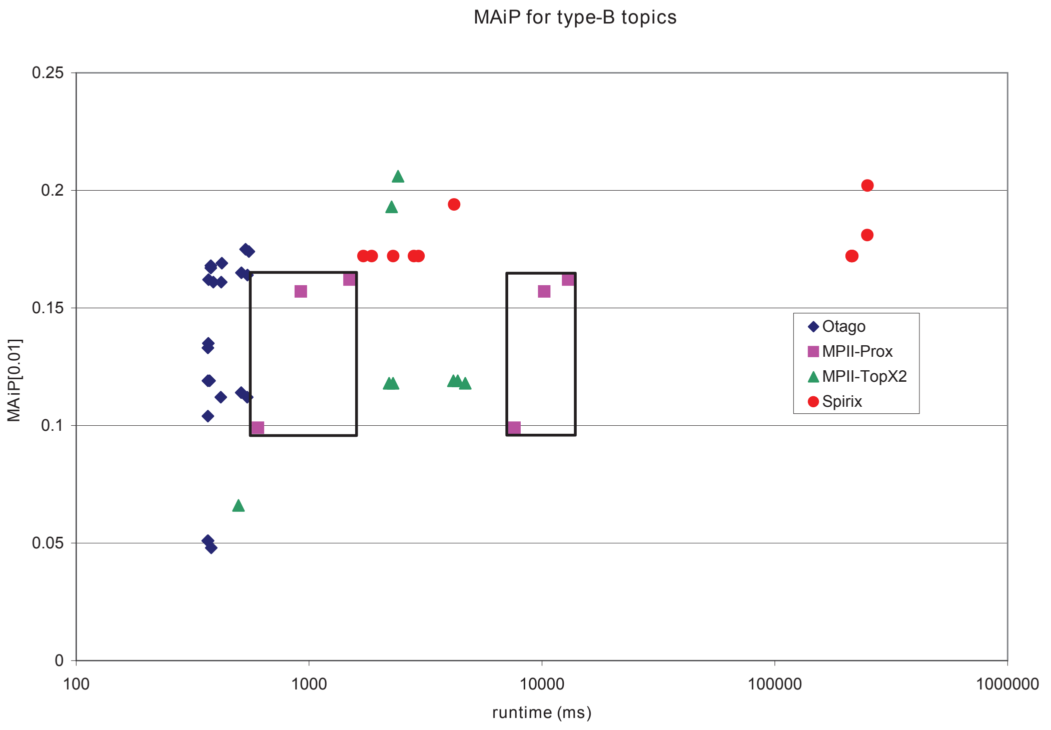


Figure 8.14: MAiP values: type B queries.

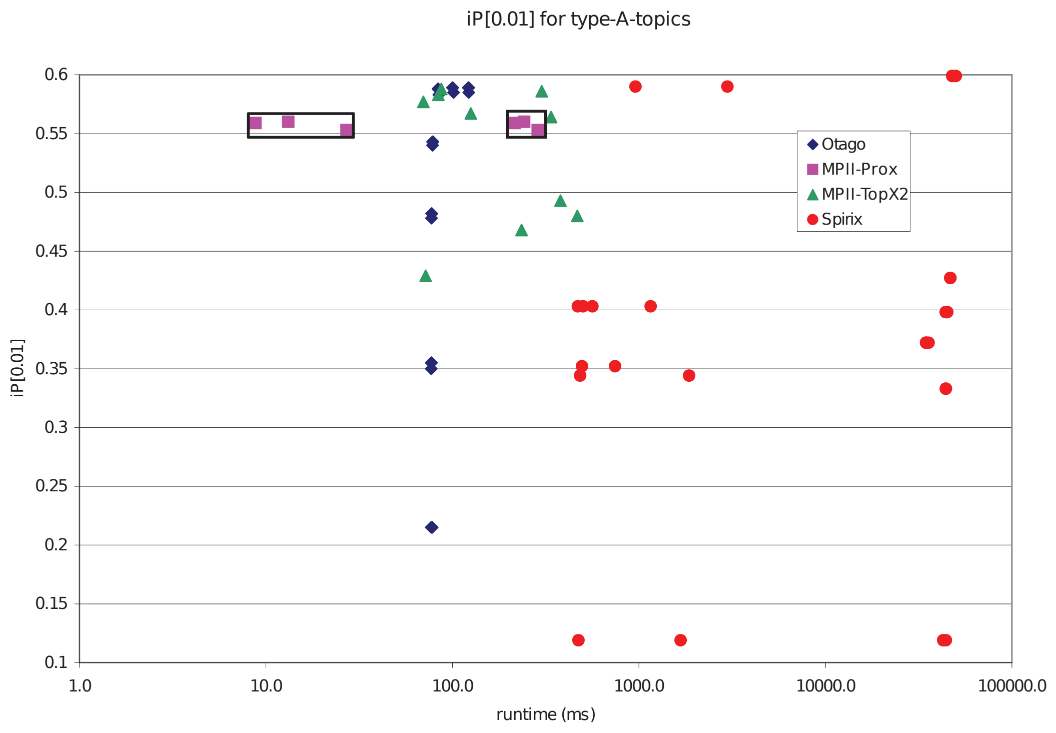


Figure 8.15: iP values: type A queries.

Spirix is very slow because it uses a distributed P2P search setting. The Otago system is fastest for type B queries since it employs highly optimized C++ code using impact layered indexes with a modified version of BM25 scores. MPII-TopX2 makes use of an inverted block structure and compresses the blocks into a customized compact binary format. Our own runs labelled MPII-Prox use merge joins with pruned indexes, tuned as described before. Squares in the left rectangle depict our hot cache runs which represent the best case where every list comes from the cache, squares in the right rectangle show cold cache runs which represent the worst case where every list lookup causes I/O costs. Our approach performs index pruning in a retrieval quality-aware manner to realize performance improvements and smaller indexes at the same time. Our best tuned index structures provide the best CPU times for type A queries among all Efficiency Track participants while still providing at least BM25 retrieval quality. Due to the number of query terms, Type B queries which consist of partly more than 100 keywords cannot be processed equally performant as type A queries: the number of pair lists to be fetched from harddisk for type B queries before they can be evaluated can easily be in the order of thousands.

8.6 Hybrid Index Structure for Efficient Text Retrieval

This section is based on our work published in [BS11]. Query processing with precomputed term pair lists can improve efficiency for some queries, but suffers from the

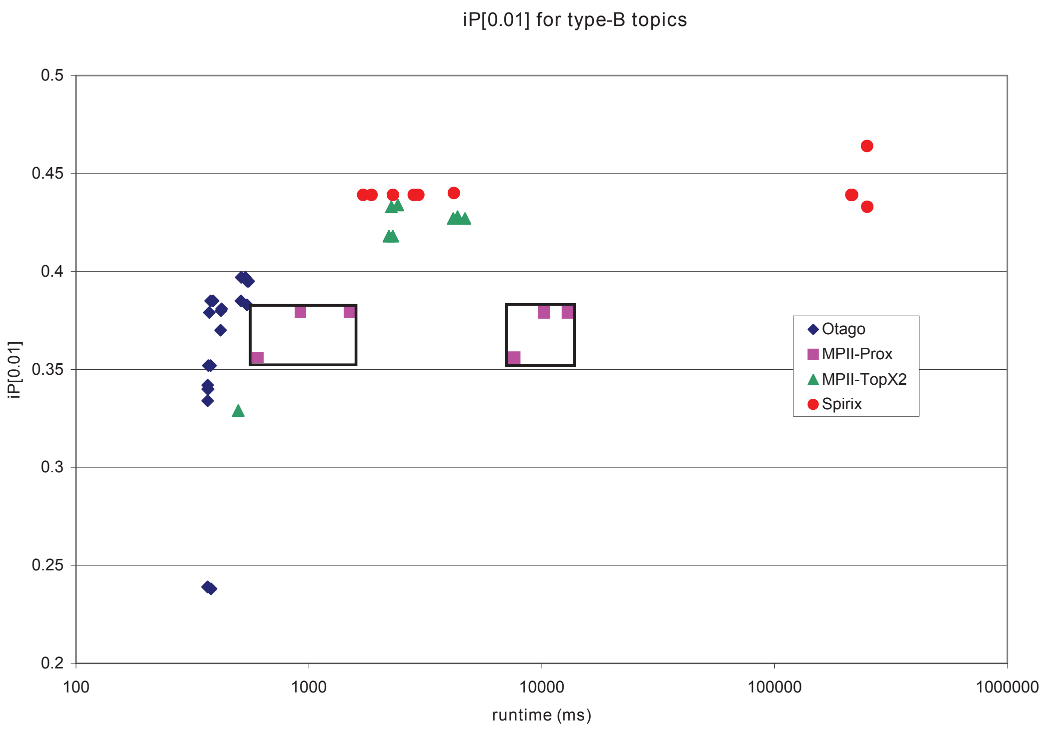


Figure 8.16: iP values: type B queries.

quadratic number of index lists that need to be read. Here, we present a novel hybrid index structure that aims at decreasing the number of index lists retrieved at query processing time, trading off a reduced number of index lists for an increased number of bytes to read.

8.6.1 Introduction

While precomputed indexes for term pairs can greatly improve performance for short queries, they are not that efficient for long queries or when lists are not available in a cache, but need to be read from disk. This disadvantage is rooted in the quadratic number of term pair lists that need to be accessed for every query. Especially with the pruning methods proposed earlier in this chapter that store only a small number of entries per term pair list, query processing time is dominated by the time to locate and open index lists. Reducing the number of index lists for processing a query can therefore significantly improve efficiency, even if more data must be read from each list. We base on and extend the index framework for TLs and CLs presented in Section 8.2. Experimental results indicated that it is enough to heuristically keep only the best few thousand entries in each list to achieve good result quality.

8.6.2 Hybrid Index Framework

To accelerate query processing, especially for medium-sized queries, it is necessary to reduce the number of lists accessed by each query. For a query with 5 terms, up to 10 CLs and 5 TLs need to be opened. The hybrid index framework that we have proposed in [BS11] can reduce this to at most 10 lists in the best case, reducing the number of lists to open by 33%. We achieve this by combining the CL for a term pair (t_1, t_2) with the TLs for t_1 and t_2, yielding an *extended combined index list* (CLExt) that now contains the best documents for both the term pair and the two single terms. We can expect that many documents will be included in two or three of the lists, so that the number of entries in the resulting CLExt will be less than the aggregated number of entries of the three source lists. Within the CLExt, we store all entries in the same format, replacing unknown scores by 0, and sort all entries by their docid.

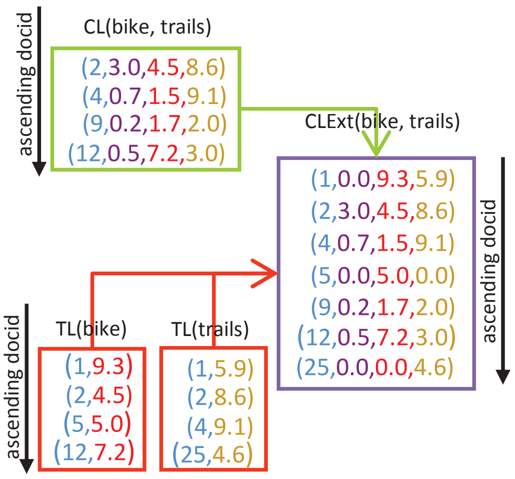


Figure 8.17: Hybrid index CLExt.

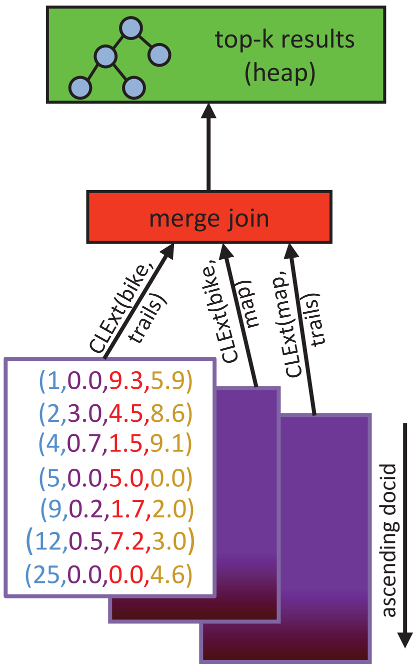


Figure 8.18: Merge join with hybrid index CLExt for query {bike, trails, map}.

Figure 8.17 shows how to combine two TLs and one CL into one CLExt and Figure 8.18 how a merge join works with the hybrid index CLExt.

At query processing time, only CLExts need to be read, reducing the number of index lists by n (for queries with n terms). For queries with 3 terms, the number of lists is only 3 compared to 6 in the existing TL+CL approach. For queries with a larger number of terms, the technique is less effective since there is still a relatively large number of CLExts to read, and information from one TL is now included in several CLExts, so some of the information read during query processing is not needed. We will see later that the break-even point is around 8 terms per query. Note that TLs need to be kept in the index for queries that consist of just a single term.

If we build a hybrid index as we just explained, the size of that index will be a lot larger than the size of the index with just TLs and CLs. While this comes as a surprise at first view, it has a simple explanation: many pairs of terms hardly occur together in the same document's text window of size W=10, so the corresponding CL is very short, but they frequently occur in isolation, so the (prefix of the) TL of each term in the index is long. The CLExt for such pairs is therefore orders of magnitude larger than the CL for the same pair. We can lower the required space for the hybrid

index type	size	build time
CL	93.2 GB	<5h
TL	1.6 GB	
CLExt	3.1 TB	70h
CLExt$_{QLog}$	131.7 GB	4.5h

Table 8.20: Index sizes and build times for full (310,0.05) indexes.

index by using additional information on how frequently pairs are used, for example from a query log. We then build CLExts only for term pairs that are used frequently enough; for all other pairs we keep the old CL scheme. This drastically reduces the size of the hybrid index, while still providing reasonable performance improvements. With the TREC GOV2 collection, generating CLExts only for term pairs that occur at least once in the AOL query log reduced the on-disk size of the CLExts from over 3TB to 131.7GB; the on-disk size of all CLs in the standard index was 93.2GB. Table 8.20 shows index sizes and build times for different index types.

8.6.3 Experimental Evaluation

For the experimental evaluation we have built full compressed indexes. We evaluated our proposed hybrid index with the GOV2 collection, using the 150 Ad Hoc topics from the TREC 2004–2006 Terabyte Track, Ad Hoc Tasks and the first 10,000 queries from the Terabyte Track, Efficiency Task (EffTrack) 2005 [CSS05] as test beds. All TLs and CLs are pruned to at most 310 entries, and entries in CLs have an *acc*-score of at least 0.05; experiments in Section 8.5.2 have shown that this is enough to yield a similar quality for top-10 documents as produced by unpruned TLs. We report average cold-cache runtimes (averaged over six independent runs) and access costs for top-10 retrieval with the original index (TL+CL) and the hybrid index with log-based pruning (TL+CLExt$_{QLog}$); file-system caches were emptied *before running each query*, which is a very conservative setting. Note that runtimes and cost are largely independent of the number of retrieved results.

For the Terabyte Track, Ad Hoc Tasks queries, using the hybrid index improved runtime from 59ms to 49ms per query over the original index; for the EffTrack queries, the improvement was even better (55ms vs 42ms per query). This clearly shows that our hybrid index can greatly improve cold-cache performance. We will now evaluate the impact for queries of different length, and the influence of log-based pruning.

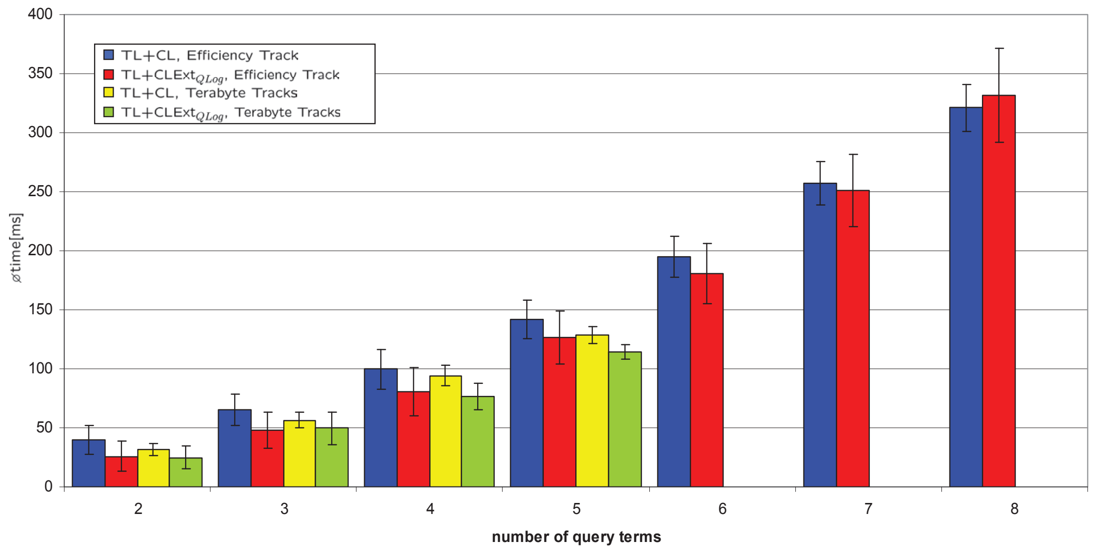


Figure 8.19: Average runtimes for Terabyte and EffTrack queries.

Figure 8.19 reports average query times for the two test beds, grouped by the number of terms per query. Improvements are best for short queries, but we see improvements up to 7 terms. The chart also indicates the standard deviations which are pretty low.

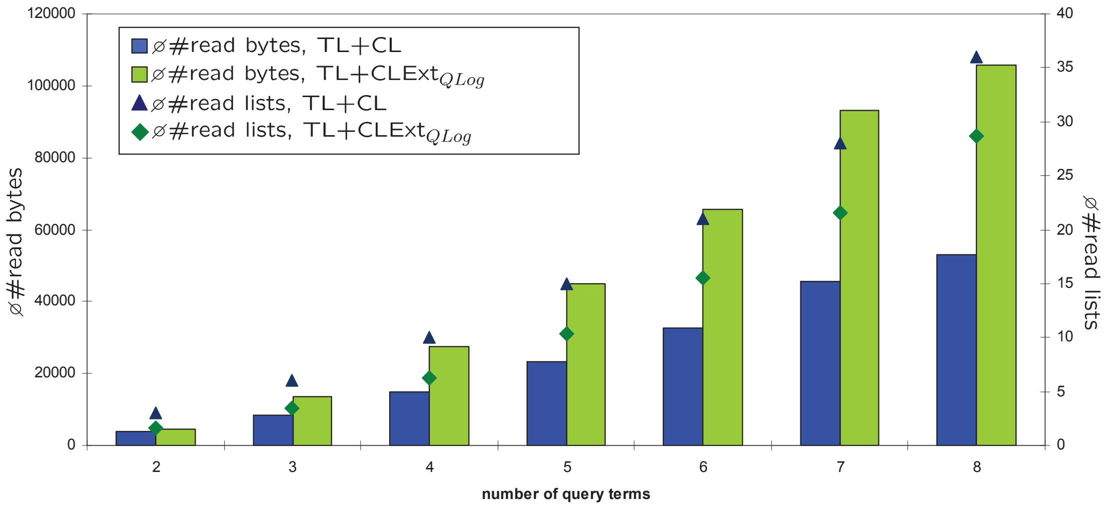


Figure 8.20: Average cost in bytes and average number of opened lists, for the EffTrack queries.

Figure 8.20 details the average number of bytes read per query for the EffTrack. The hybrid index reads up to twice as many bytes from disk, but is (as we saw before) still faster because it needs to open fewer lists (also depicted in this figure by triangles and diamonds).

Figure 8.21 shows the influence of log-based pruning on runtime. We computed,

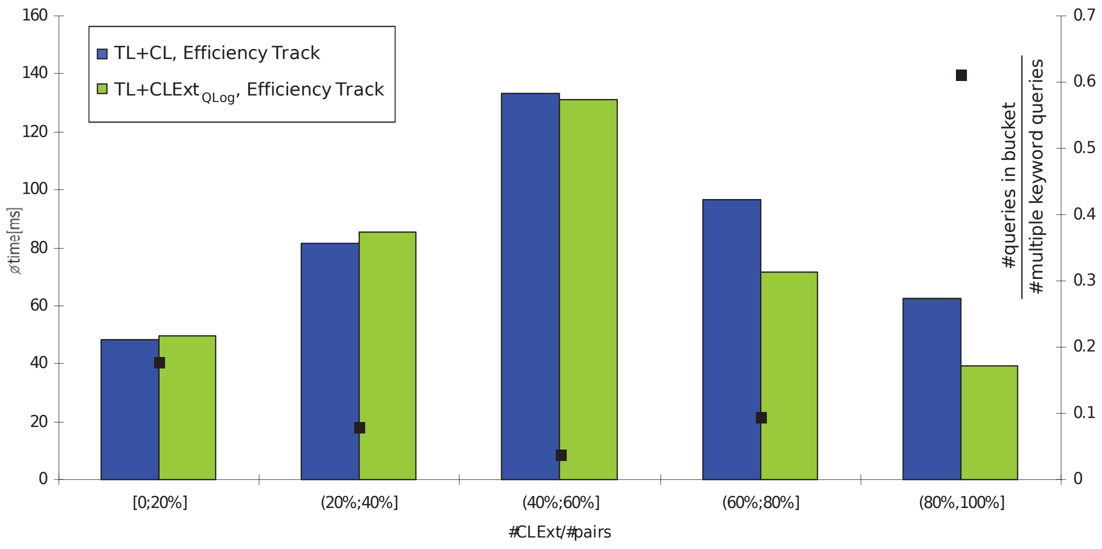


Figure 8.21: Effect of query term pair coverage in the AOL query log on runtime, for the EffTrack queries.

for each EffTrack query, the fraction of term pairs covered in the log, and grouped queries into five buckets from low coverage (0%-20%) to high coverage (80%-100%). Our method gives benefit only for queries with a term pair coverage of at least 60%; however, these are the most frequent queries in this load (indicated by the black dots). For the remaining queries, our method does not create a performance penalty.

8.7 Conclusion

We clearly demonstrated that indexing terms and term pairs, together with tunable list pruning, is a viable method to improve either result quality or, providing a similar quality as pure term indexes, processing performance. Results with effectiveness-oriented indexes are comparable to the best results using unpruned indexes and efficiency-oriented index configurations yield almost one order of magnitude performance gain compared to a state-of-the-art top-k algorithm. We have demonstrated that our hybrid index structure significantly improves cold-cache query processing times of almost 25% on standard benchmark queries from TREC Terabyte and Efficiency Tracks by decreasing the number of fetched index lists, at the price of reading more from each list. The highest improvements are achieved for short queries.

Chapter 9

Conclusion and Outlook

9.1 Conclusion

In the presence of growing data, the need for efficient query processing under result quality and index size control becomes more and more a challenge to search engines. This work has shown how to use proximity scores to make query processing effective and efficient with focus on either of the optimization goals.

This thesis made the following important contributions:

- We have presented a comprehensive comparative analysis of proximity score models and a rigorous analysis of the potential of phrases and have adapted a leading proximity score model for XML data.
- We have discussed the feasibility of all presented proximity score models for top-k query processing and have presented a novel index combining a content and proximity score that helps to accelerate top-k query processing and improves result quality.
- We have presented a novel, distributed index tuning framework for term and term pair index lists that optimizes pruning parameters by means of well-defined optimization criteria under disk space constraints. Indexes can be tuned with emphasis on efficiency or effectiveness: the resulting indexes yield fast processing at high result quality.
- We have shown that pruned index lists processed with a merge join outperform top-k query processing with unpruned lists at a high result quality.
- Moreover, we have presented a hybrid index structure for improved cold cache run times.

9.2 Outlook

There are still some interesting remaining open challenges that deserve future attention.

As our index tuning framework only uses term and term pair lists (e.g., combined lists), a possible extension would be to precompute (selected) term n-tuple lists for $n > 2$. Furthermore, extending index lists with more features could generate better retrieval quality at lower query processing costs: some query-independent features like the PageRank score may be easily integrated into term indexes (cf. Section 7.3.1) whereas other features may pose real challenges for integration into the existing framework and ask for novel index structures.

The evaluation and integration of further pruning methods such as the document-centric pruning by Büttcher and Clarke (see Section 6.2.3 for more details) into our index tuning framework may be worth investigation.

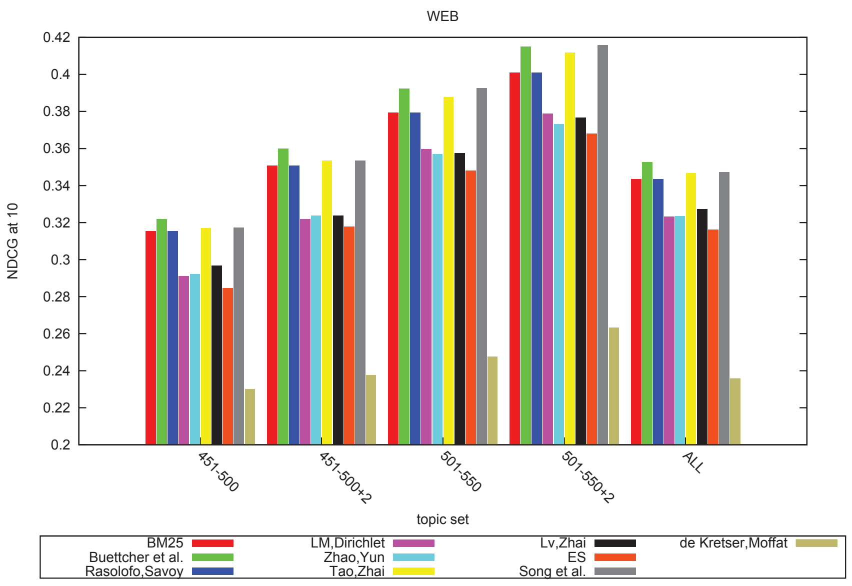
Possible improvements as to index construction, index tuning, and index maintenance are as follows: improving the construction of the final index by reducing the number of temporary index entries, and improving the estimation stage which currently needs to parse the complete document collection would reduce the time required to build up an index and to find optimal pruning parameters. Further consideration of the impact of pruned indexes on cache effectiveness and a careful index layout that groups frequently co-occurring lists close to each other will be a key extension for further improving processing time with cold caches.

Index maintenance such as supporting index updates would be especially beneficial for dynamic data such as email collections. So far the index has to be completely rebuilt and the optimization process is repeated completely if new documents are added to a collection. For dynamic data, it is required to develop means to incrementally add new documents and regularly optimize tuning parameters without the need to completely rebuild the index.

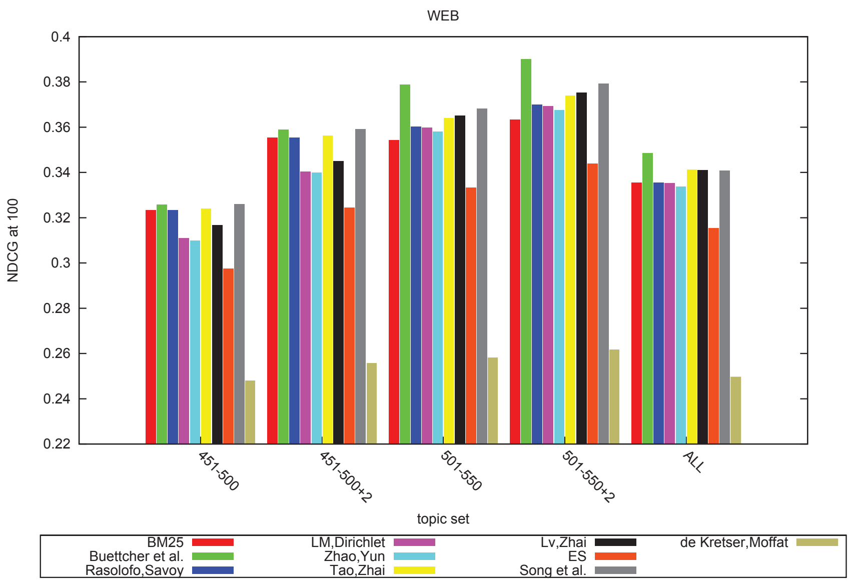
Finally, our hybrid index structure may be improved in the following ways: our hybrid index structure significantly improves cold-cache query processing times by decreasing the number of fetched index lists, at the price of reading more from each list. As the highest performance improvements are achieved for short queries, future work may concentrate on improving performance for long queries, for example by precomputing lists for frequently used phrases or removing non-important pair lists. Extending our index tuning framework to optimize pruning parameters for our hybrid index structure would certainly enrich our framework.

Appendix A

Retrieval Quality and Sensitivity

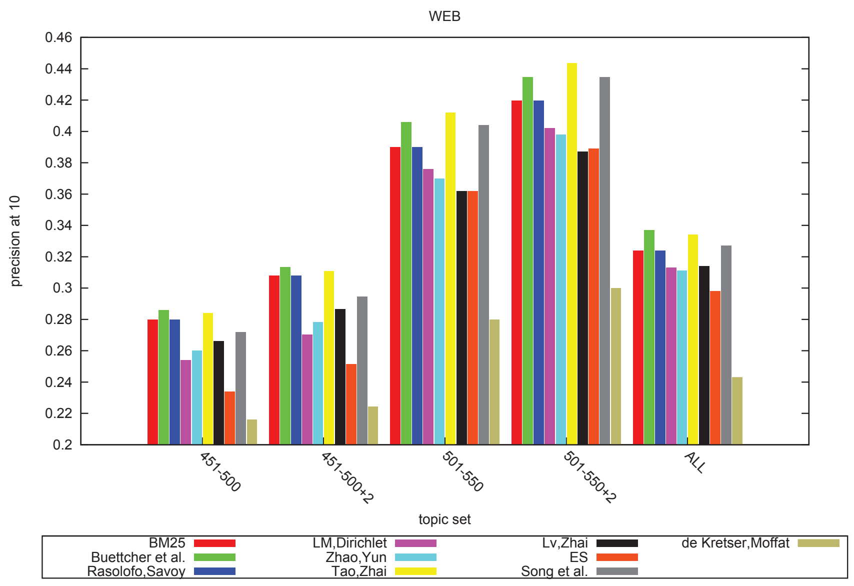


(a) Web Track (WT10g): best NDCG@10 values per scoring model

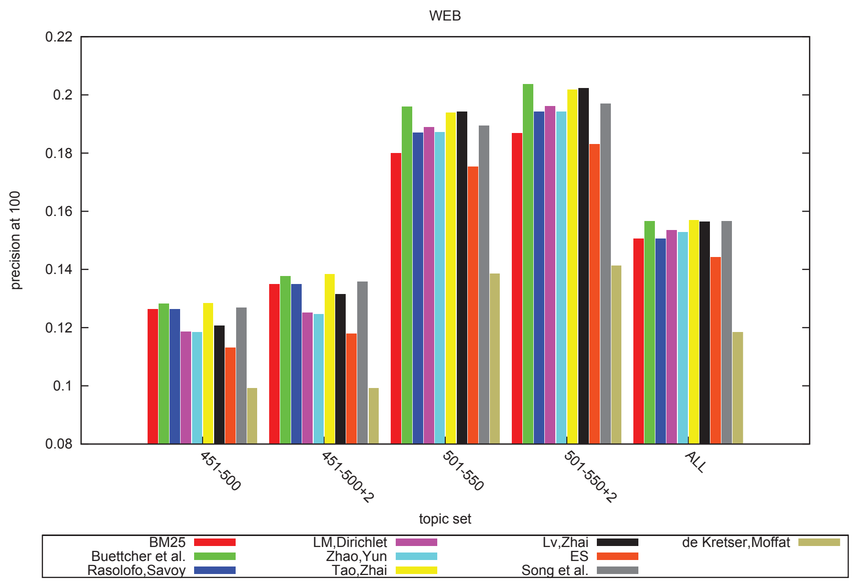


(b) Web Track (WT10g): best NDCG@100 values for per scoring model

Figure A.1: Web Tracks test beds (WT10g): best NDCG values



(a) Web Track (WT10g): best P@10 values per scoring model



(b) Web Track (WT10g): best P@100 values for per scoring model

Figure A.2: Web Tracks (WT10g): best precision values

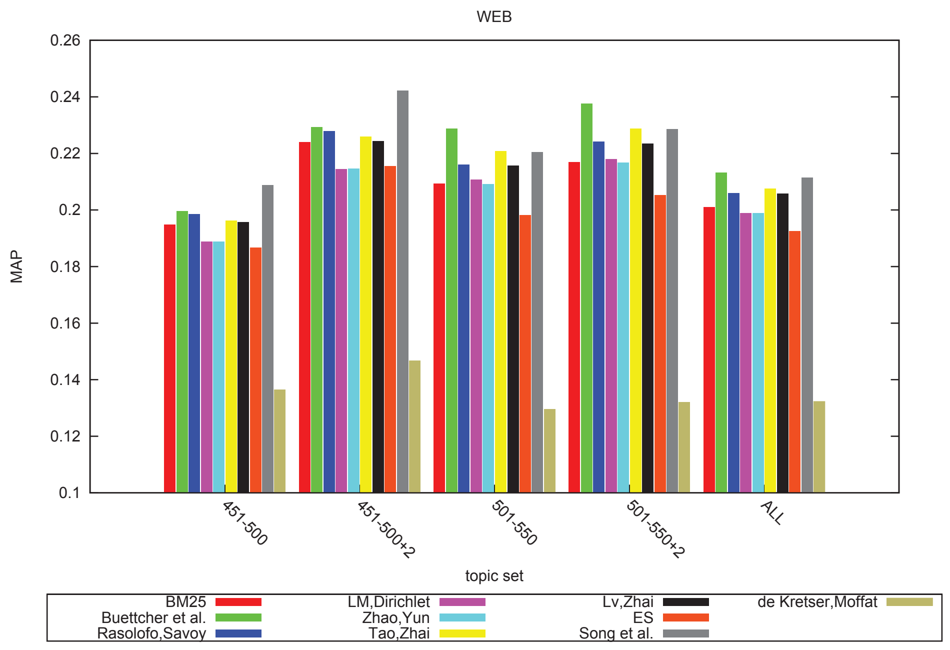
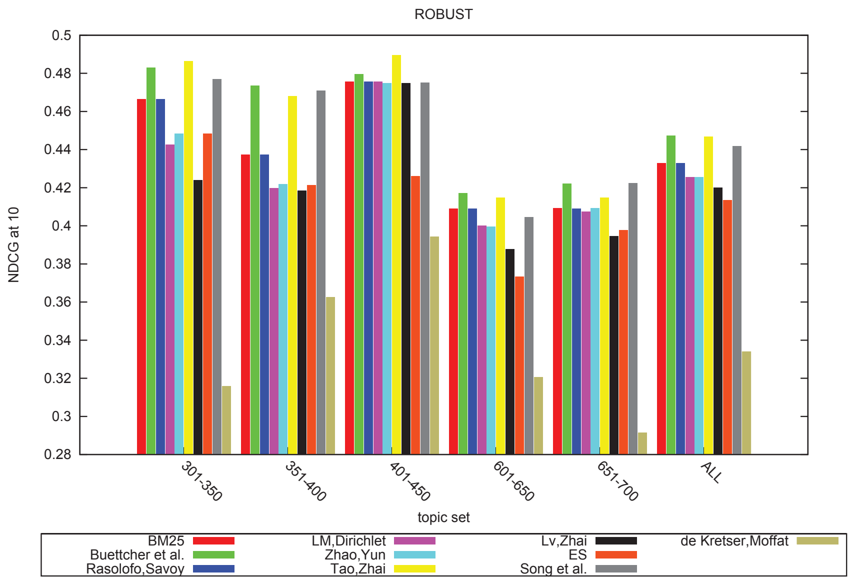
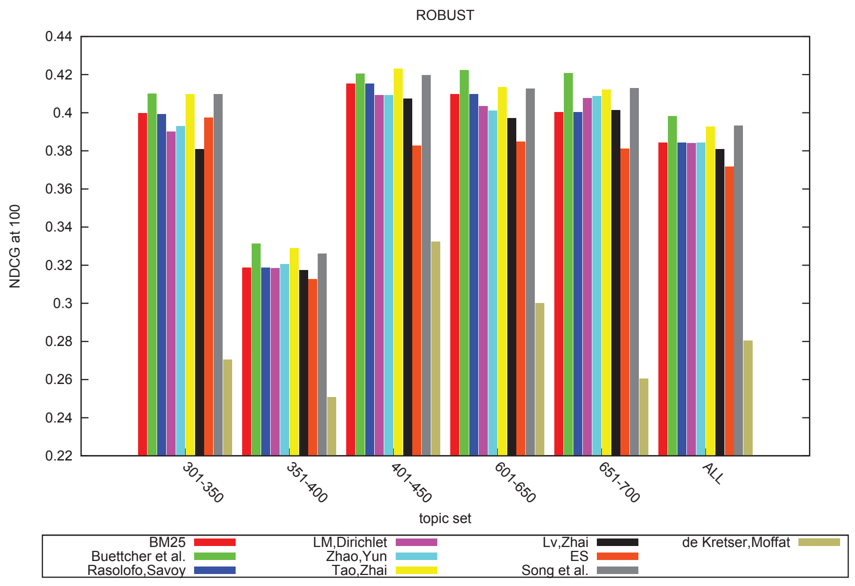


Figure A.3: Web Track (WT10g): best MAP values for each scoring model

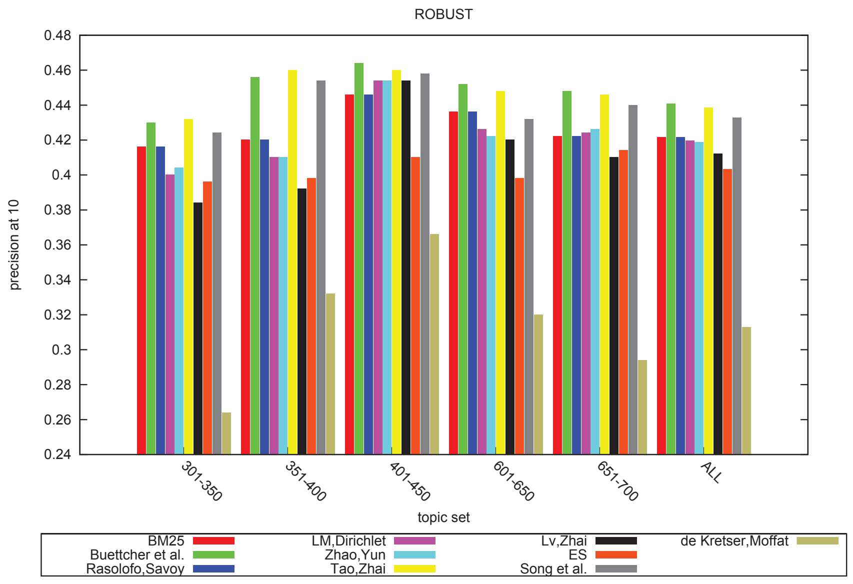


(a) Robust Track: best NDCG@10 values for each scoring model

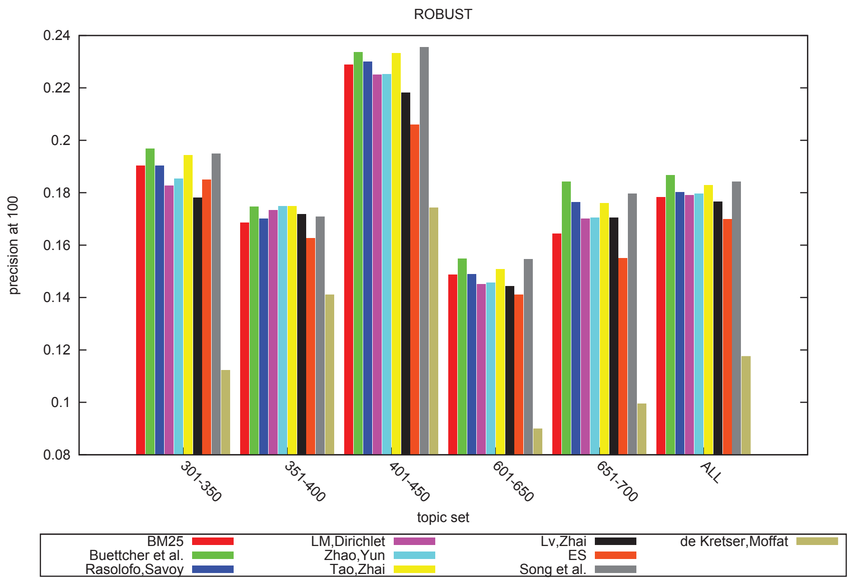


(b) Robust Track: best NDCG@100 values for each scoring model

Figure A.4: Robust Track: best NDCG values



(a) Robust Track: best P@10 values per scoring model



(b) Robust Track: best P@100 values per scoring model

Figure A.5: Robust Track: best precision values

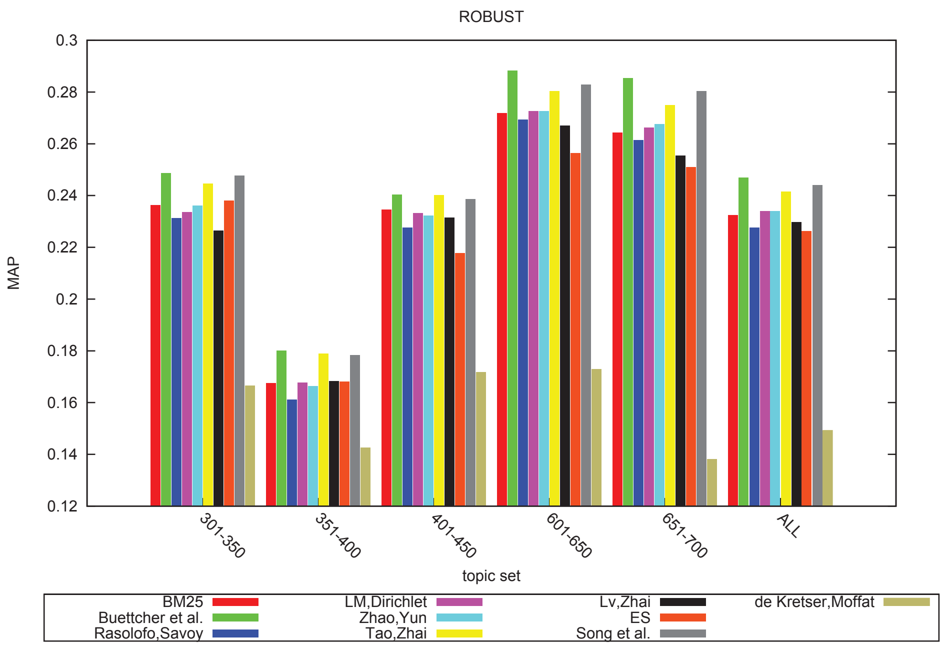
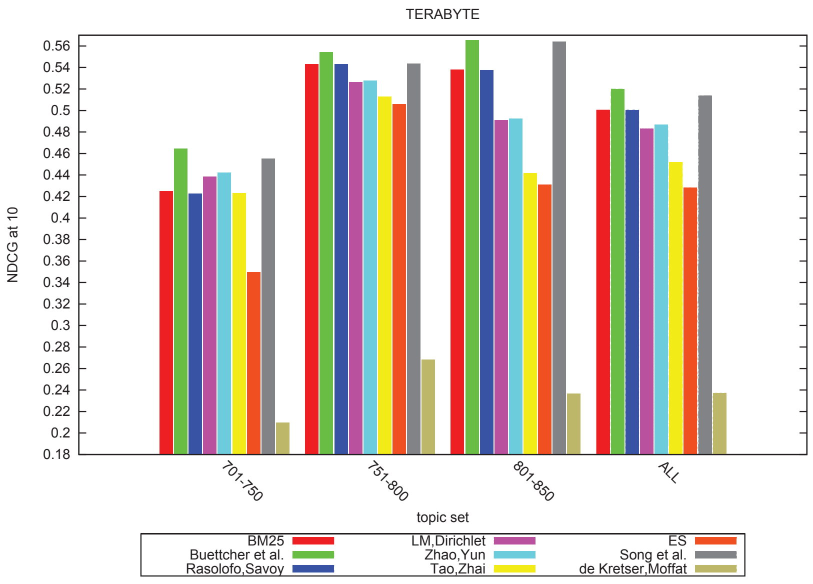
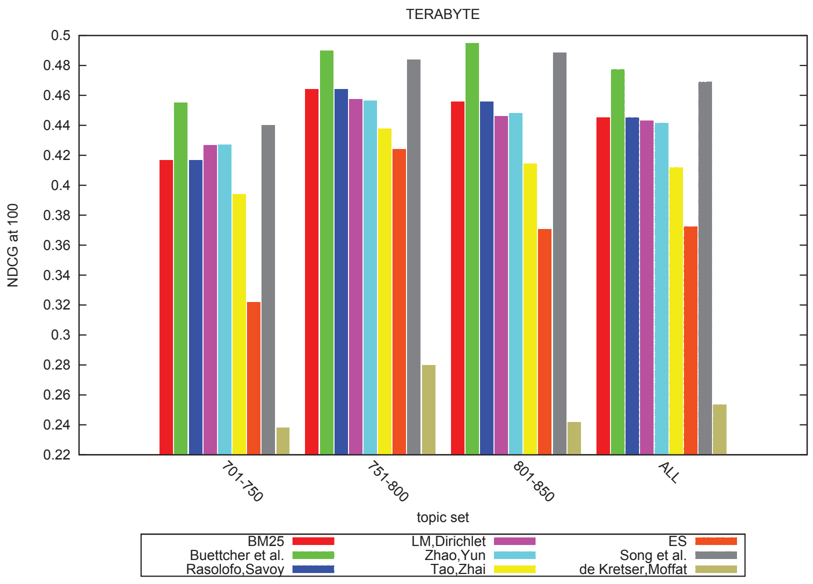


Figure A.6: Robust Track: best MAP values for each scoring model

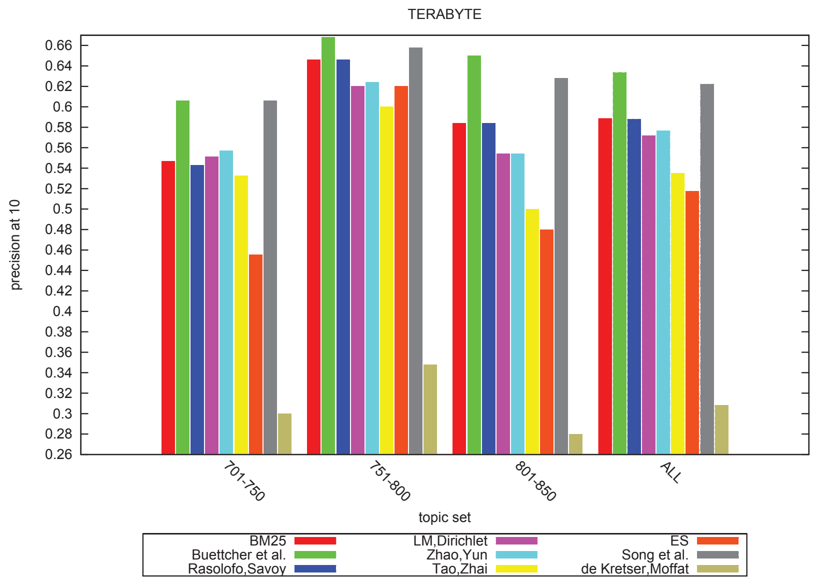


(a) Terabyte Track: best NDCG@10 values per scoring model

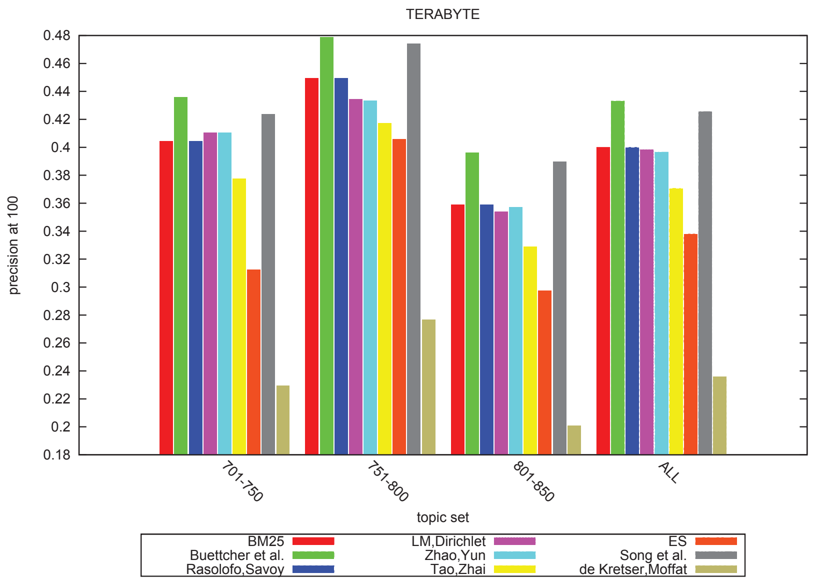


(b) Terabyte Track: best NDCG@100 values per scoring model

Figure A.7: Terabyte Track: best NDCG values



(a) Terabyte Track: best P@10 values per scoring model



(b) Terabyte Track: best P@100 values per scoring model

Figure A.8: Terabyte Track: best precision values

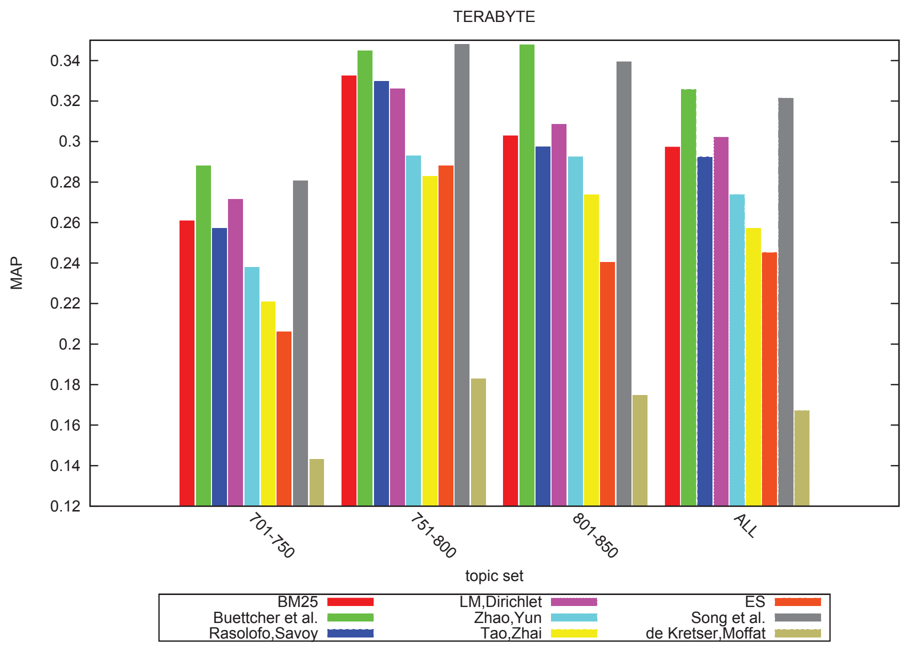
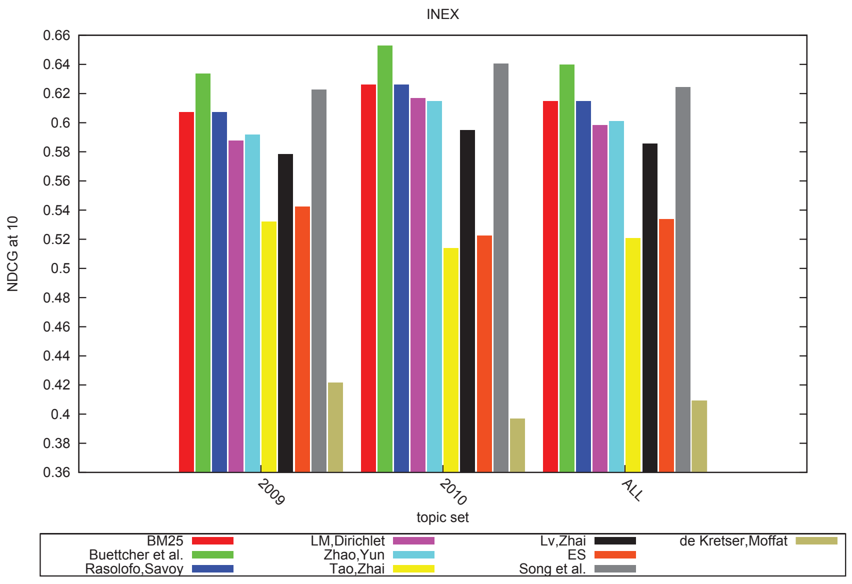
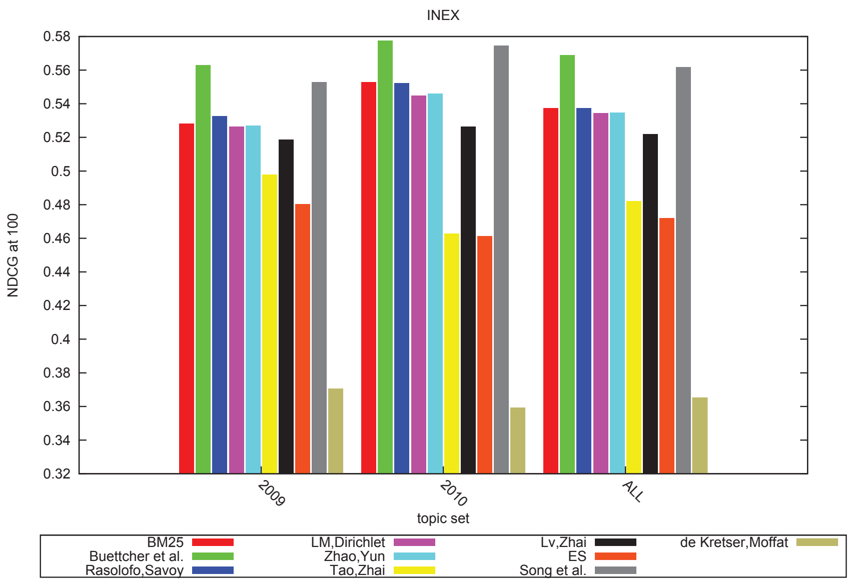


Figure A.9: Terabyte Track: best MAP values for each scoring model

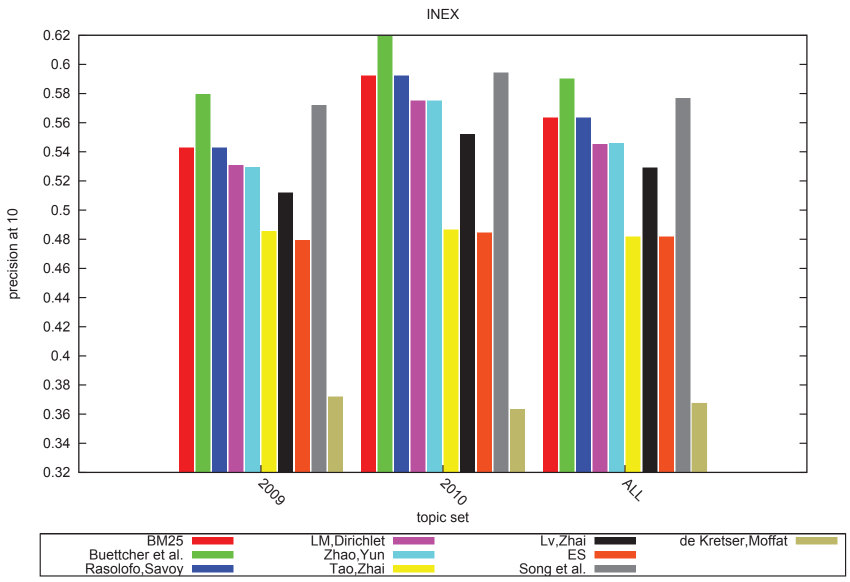


(a) INEX: best NDCG@10 values per scoring model

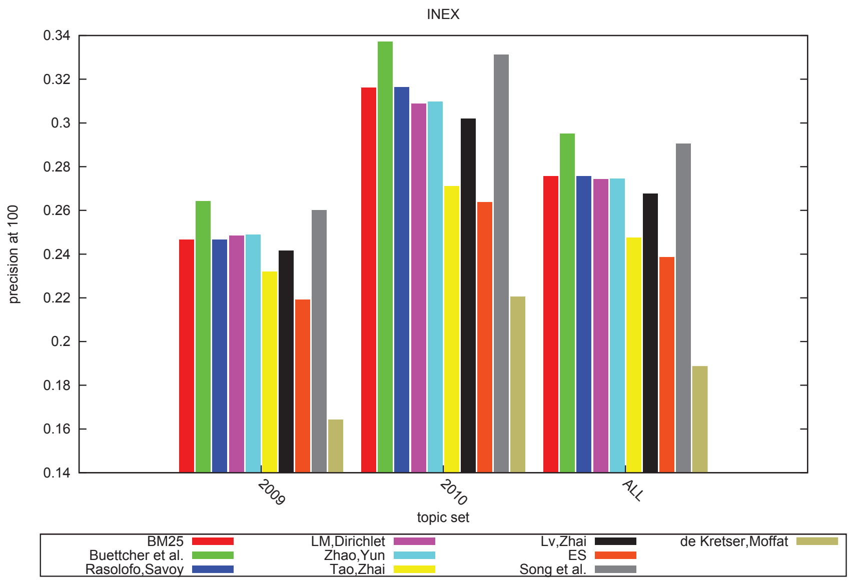


(b) INEX: best NDCG@100 values per scoring model

Figure A.10: INEX: best NDCG values



(a) INEX: best P@10 values per scoring model



(b) INEX: best P@100 values per scoring model

Figure A.11: INEX: best precision values

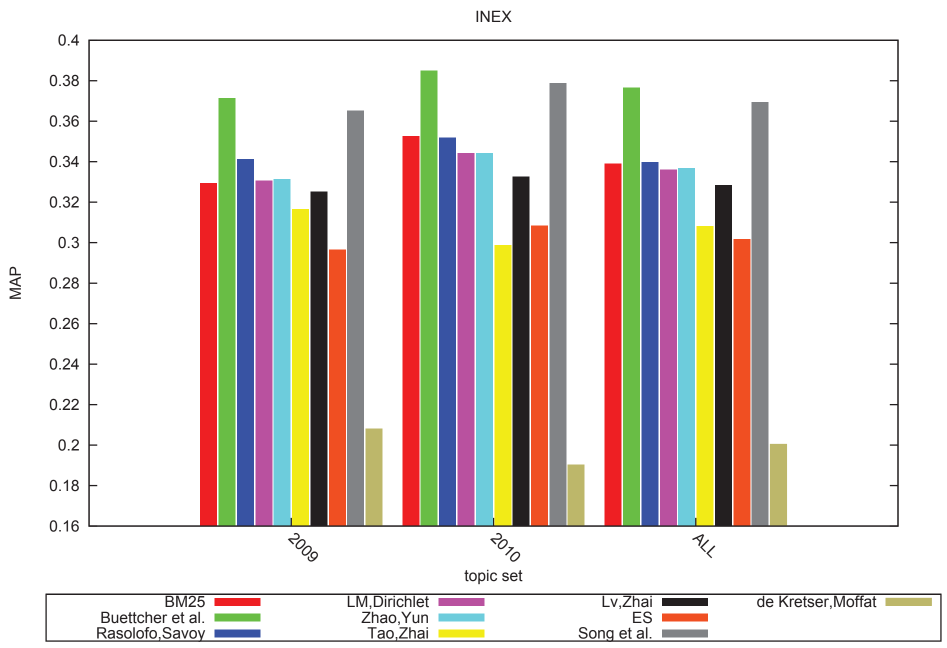
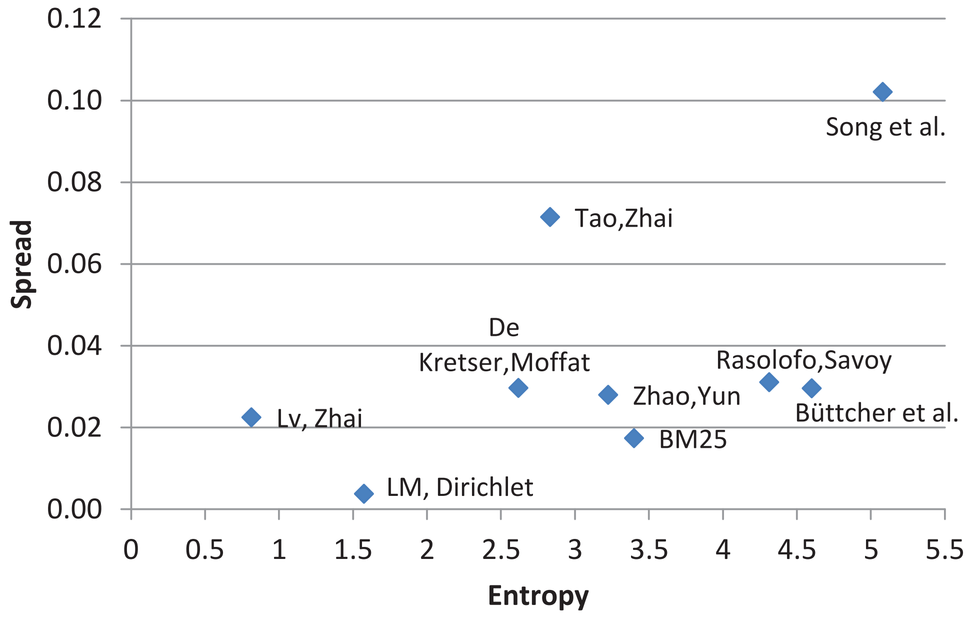
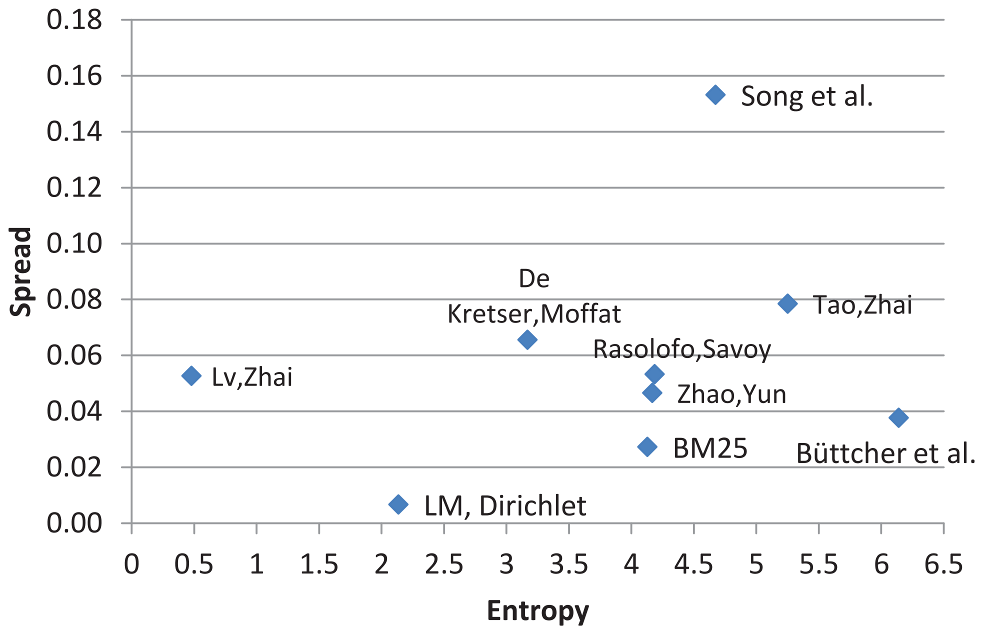


Figure A.12: INEX: best MAP values for each scoring model

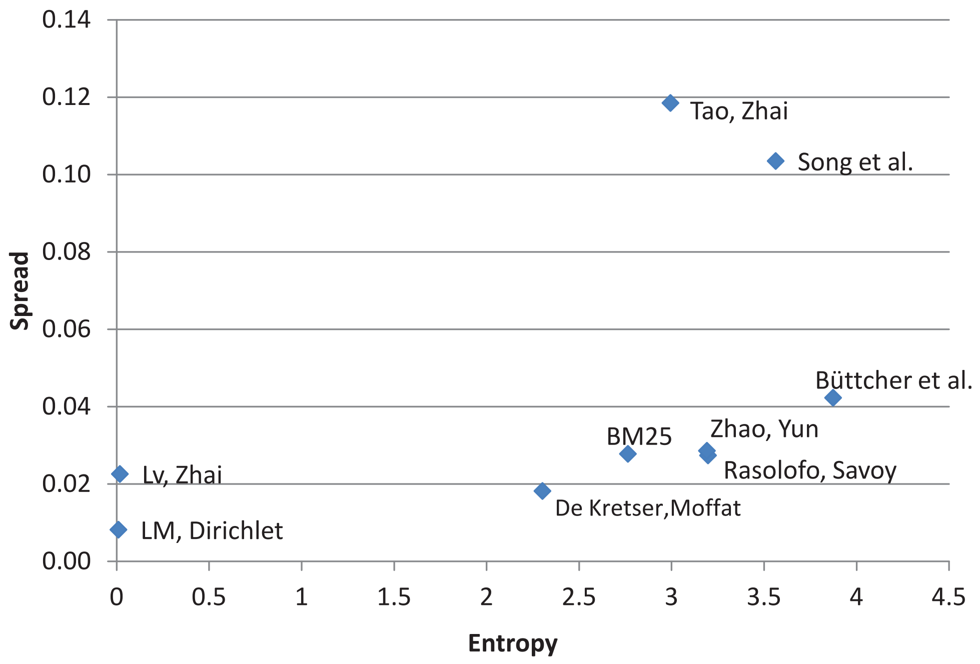


(a) WEB: sensitivity of scoring models for MAP

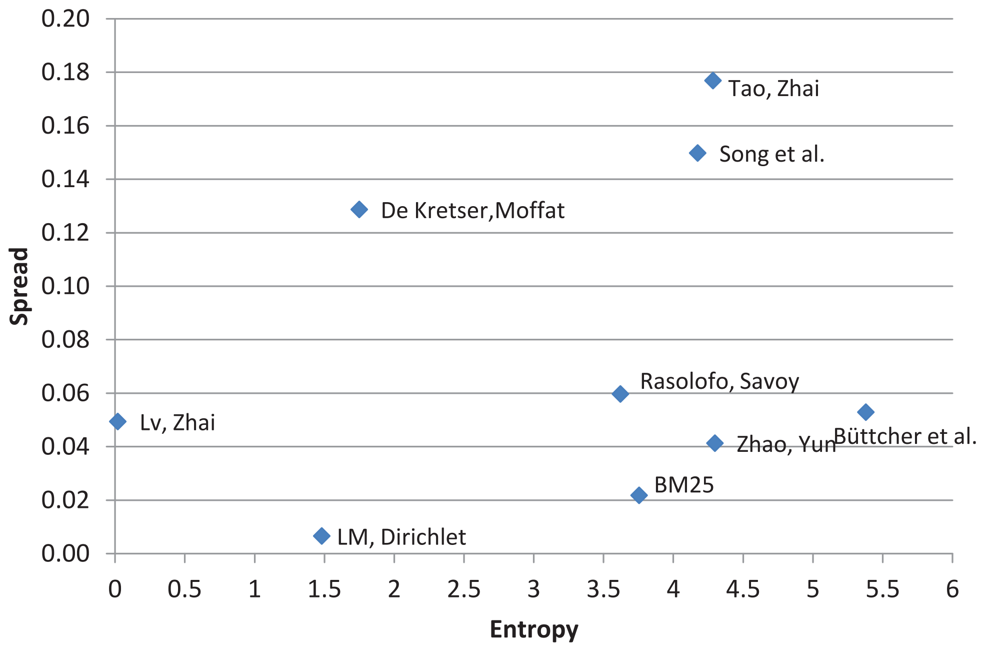


(b) WEB: sensitivity of scoring models for NDCG@10

Figure A.13: WEB: sensitivity of scoring models

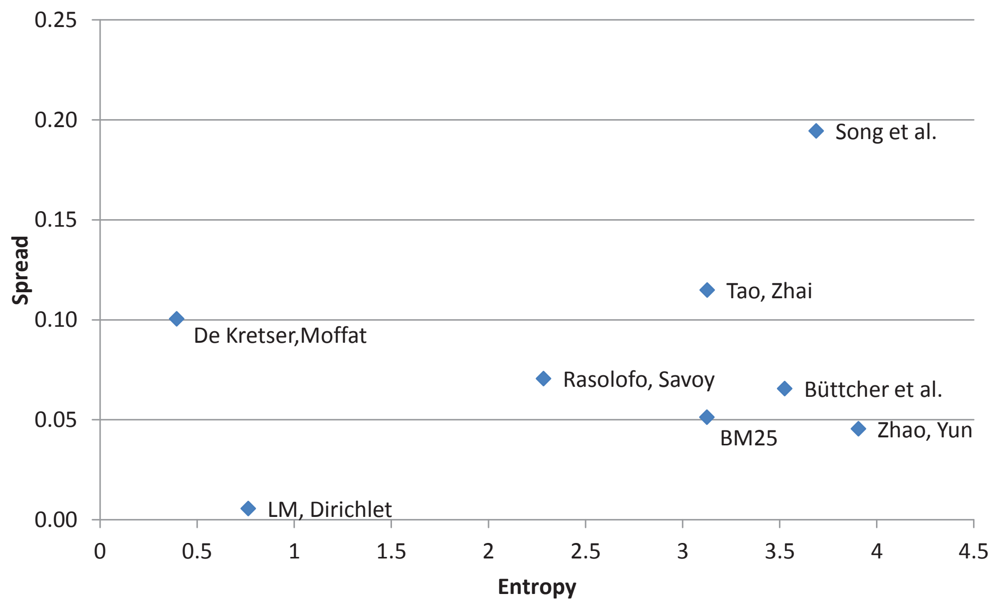


(a) ROBUST: sensitivity of scoring models for MAP

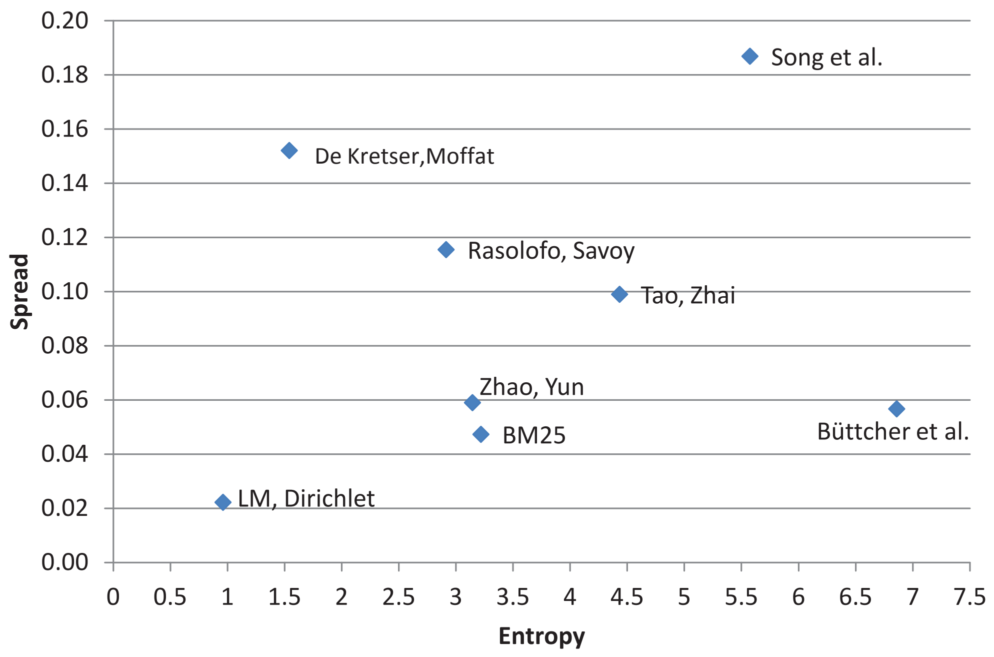


(b) ROBUST: sensitivity of scoring models for NDCG@10

Figure A.14: ROBUST: sensitivity of scoring models

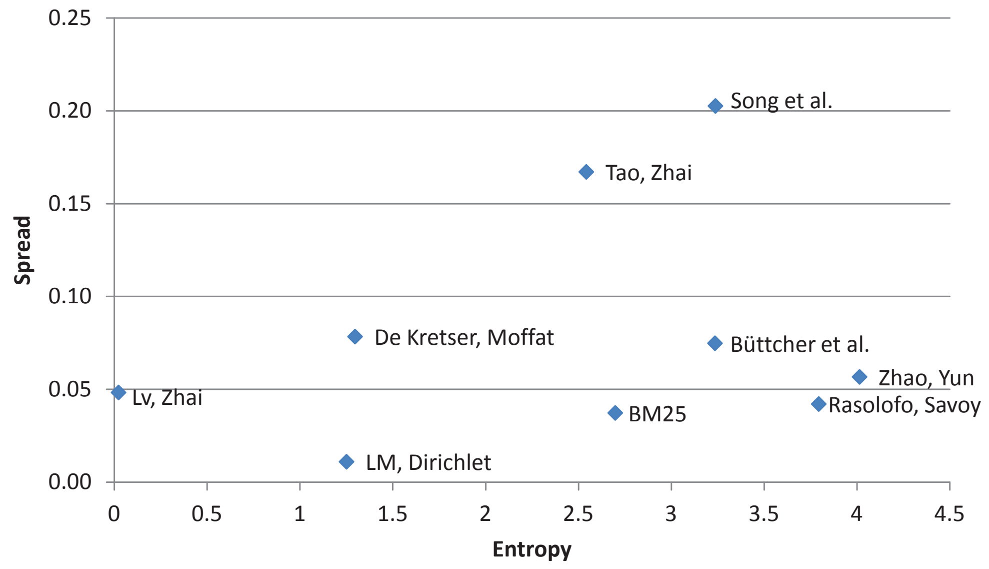


(a) TERABYTE: sensitivity of scoring models for MAP

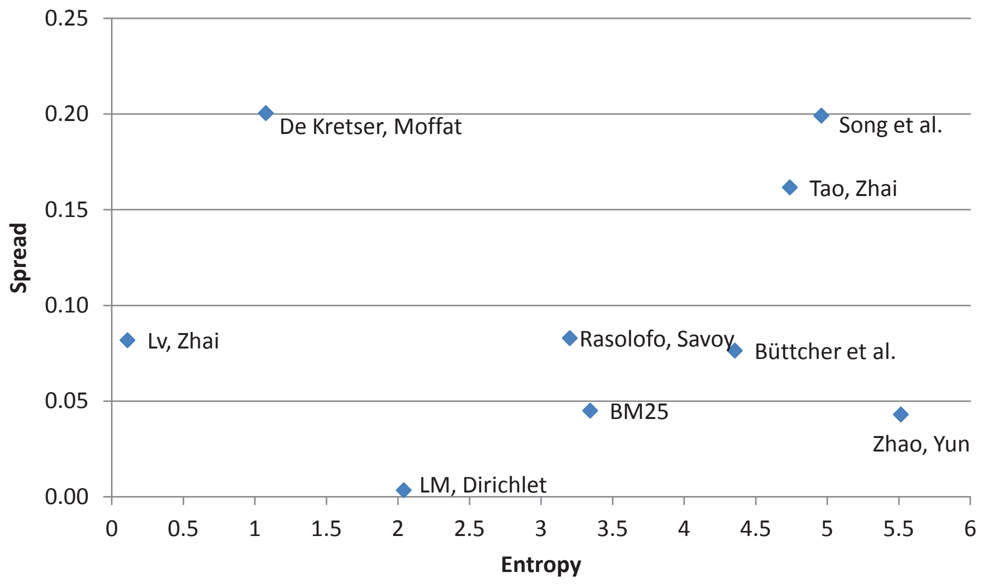


(b) TERABYTE: sensitivity of scoring models for NDCG@10

Figure A.15: TERABYTE: sensitivity of scoring models



(a) INEX: sensitivity of scoring models for MAP



(b) INEX: sensitivity of scoring models for NDCG@10

Figure A.16: INEX: sensitivity of scoring models

Appendix B

TREC

num: title	num: title
701: U.S. oil industry history	726: Hubble telescope repairs
702: Pearl farming	727: Church arson
703: U.S. against International Criminal Court	728: whales save endangered
704: Green party political views	729: Whistle blower department of defense
705: Iraq foreign debt reduction	730: Gastric bypass complications
706: Controlling type II diabetes	731: Kurds history
707: Aspirin cancer prevention	732: U.S. cheese production
708: Decorative slate sources	733: Airline overbooking
709: Horse racing jockey weight	734: Recycling successes
710: Prostate cancer treatments	735: Afghan women condition
711: Train station security measures	736: location BSE infections
712: Pyramid scheme	737: Enron California energy crisis
713: Chesapeake Bay Maryland clean	738: Anthrax hoaxes
714: License restrictions older drivers	739: Habitat for Humanity
715: Schizophrenia drugs	740: regulate assisted living Maryland
716: Spammer arrest sue	741: Artificial Intelligence
717: Gifted talented student programs	742: hedge funds fraud protection
718: Controlling acid rain	743: Freighter ship registration
719: Cruise ship damage sea life	744: Counterfeit ID punishments
720: Federal welfare reform	745: Doomsday cults
721: Census data applications	746: Outsource job India
722: Iran terrorism	747: Library computer oversight
723: Executive privilege	748: Nuclear reactor types
724: Iran Contra	749: Puerto Rico state
725: Low white blood cell count	750: John Edwards womens issues

Table B.1: TREC 2004 Terabyte Track, Ad Hoc Task topics.

num: title	num: title
751: Scrabble Players	776: Magnet schools success
752: Dam removal	777: hybrid alternative fuel cars
753: bullying prevention programs	778: golden ratio
754: domestic adoption laws	779: Javelinas range and description
755: Scottish Highland Games	780: Arable land
756: Volcanic Activity	781: Squirrel control and protections
757: Murals	782: Orange varieties seasons
758: Embryonic stem cells	783: school mercury poisoning
759: civil war battle reenactments	784: mersenne primes
760: american muslim mosques schools	785: Ivory-billed woodpecker
761: Problems of Hmong Immigrants	786: Yew trees
762: History of Physicians in America	787: Sunflower Cultivation
763: Hunting deaths	788: Reverse mortgages
764: Increase mass transit use	789: abandoned mine reclamation
765: ephedra ma huang deaths	790: women's rights in Saudi Arabia
766: diamond smuggling	791: Gullah geechee language culture
767: Pharmacist License requirements	792: Social Security means test
768: Women in state legislatures	793: Bagpipe Bands
769: Kroll Associates Employees	794: pet therapy
770: Kyrgyzstan-United States relations	795: notable cocker spaniels
771: deformed leopard frogs	796: Blue Grass Music Festival history
772: flag display rules	797: reintroduction of gray wolves
773: Pennsylvania slot machine gambling	798: Massachusetts textile mills
774: Causes of Homelessness	799: Animals in Alzheimer's research
775: Commercial candy makers	800: Ovarian Cancer Treatment

Table B.2: TREC 2005 Terabyte Track, Ad Hoc Task topics.

num: title	num: title
801: Kudzu Pueraria lobata	826: Florida Seminole Indians
802: Volcano eruptions global temperature	827: Hidden Markov Modeling HMM
803: May Day	828: secret shoppers
804: ban on human cloning	829: Spanish Civil War support
805: Identity Theft Passport	830: model railroads
806: Doctors Without Borders	831: Dulles Airport security
807: Sugar tariff-rate quotas	832: labor union activity
808: North Korean Counterfeiting	833: Iceland government
809: wetlands wastewater treatment	834: Global positioning system earthquakes
810: timeshare resales	835: Big Dig pork
811: handwriting recognition	836: illegal immigrant wages
812: total knee replacement surgery	837: Eskimo History
813: Atlantic Intracoastal Waterway	838: urban suburban coyotes
814: Johnstown flood	839: textile dyeing techniques
815: Coast Guard rescues	840: Geysers
816: USAID assistance to Galapagos	841: camel North America
817: sports stadium naming rights	842: David McCullough
818: Chaco Culture National Park	843: Pol Pot
819: 1890 Census	844: segmental duplications
820: imported fire ants	845: New Jersey tomato
821: Internet work-at-home scams	846: heredity and obesity
822: Custer's Last Stand	847: Portugal World War II
823: Continuing care retirement communities	848: radio station call letters
824: Civil Air Patrol	849: Scalable Vector Graphics
825: National Guard Involvement in Iraq	850: Mississippi River flood

Table B.3: TREC 2006 Terabyte Track, Ad Hoc Task topics.

topic number: query	topic number: query
1:obama family tree	26:lower heart rate
2:french lick resort and casino	27:starbucks
3:getting organized	28:inuyasha
4:toilet	29:ps 2 games
5:mitchell college	30:diabetes education
6:kcs	31:atari
7:air travel information	32:website design hosting
8:appraisals	33:elliptical trainer
9:used car parts	34:cell phones
10:cheap internet	35:hoboken
11:gmat prep classes	36:gps
12:djs	37:pampered chef
13:map	38:dogs for adoption
14:dinosaurs	39:disneyland hotel
15:espn sports	40:michworks
16:arizona game and fish	41:orange county convention center
17:poker tournaments	42:the music man
18:wedding budget calculator	43:the secret garden
19:the current	44:map of the united states
20:defender	45:solar panels
21:volvo	46:alexian brothers hospital
22:rick warren	47:indexed annuity
23:yahoo	48:wilson antenna
24:diversity	49:flame designs
25:euclid	50:dog heat

Table B.4: TREC 2009 Web Track, Ad Hoc Task topics.

topic number: query	topic number: query
51:horse hooves	76:raised gardens
52:avp	77:bobcat
53:discovery channel store	78:dieting
54:president of the united states	79:voyager
55:iron	80:keyboard reviews
56:uss yorktown charleston sc	81:afghanistan
57:ct jobs	82:joints
58:penguins	83:memory
59:how to build a fence	84:continental plates
60:bellevue	85:milwaukee journal sentinel
61:worm	86:bart sf
62:texas border patrol	87:who invented music
63:flushing	88:forearm pain
64:moths	89:ocd
65:korean language	90:mgb
66:income tax return online	91:er tv show
67:vldl levels	92:the wall
68:pvc	93:raffles
69:sewing instructions	94:titan
70:to be or not to be that is the question	95:earn money at home[1]
71:living in india	96:rice
72:the sun	97:south africa
73:neil young	98:sat
74:kiwi	99:satellite
75:tornadoes	100:rincon puerto rico[1]

Table B.5: TREC 2010 Web Track, Ad Hoc Task topics ([1] indicates non-assessed topics).

Appendix C

INEX

topic_id	title
289	emperor ”Napoleon I” Polish
290	”genetic algorithm”
291	Olympian god or goddess
292	Italian Flemish painting Renaissance -French -German
293	wifi security encryption
294	user interface design usability guidelines
295	software intellectual property patent license
296	”Borussia Dortmund” + European Championship Intercontinental Cup
297	”cool jazz” ”West coast” musician
298	George Orwell life books essays 1984 Eric Arthur Blair Animal Farm
300	Airbus A380 ordered
301	Algebraic Vector Space Model generalized vector space model Latent Semantic Indexing Topic-Based vector space model extended Boolean model enhanced topic based Salton SMART
302	”web services” security standards
303	fractal applications -art
304	allergy treatments
305	”revision control system”
306	theories Studies genre classification structuralist Plato forms Aristotle forms
308	wedding traditions and customs
309	”Ken Doherty” finals tournament
310	Novikov self-consistency principle and time travel
311	global warming cause and effects
312	recessive genes and hereditary disease or genetic disorder
313	”immanuel kant” ”moral philosophy” ”categorical imperative”
314	food additive toxin carcinogen ”E number”
315	spider hunting insect
316	differents disciplines and movements for gymnastics sport
317	tourism paris visit museum cathedral

Table C.1: INEX CO topics with relevance assessments, Ad Hoc Track 2006, part 1

topic_id	title
318	the atlantic ocean islands and the slave trade
319	”northern lights” ”polar lights” ”aurora borealis” ”solar wind” ”magnetic field” earth
320	paris transport ”Gare de Lyon” ”Gare du Nord”
321	buildings designed Antoni Gaudi Barcelona architect
322	castles kasteel in the netherlands
323	founder ikea
324	composition of planet rings
325	”Cirque du Soleil” shows
326	Scotland tourism
327	cloning animals accepted ”United States of America”
328	NBA European basketball player
329	”national dress” +Scottish
330	”nobel prize” laureate physics dutch Netherlands
331	figure tulips
332	NCAA basketball tournament ”march madness”
333	steve wozniak steve jobs
334	”Silk Road” China
335	prepare acorn eat
336	species of monotreme
337	security algorithms in computer networks
338	high blood pressure effect
339	Toy Story
340	Reinforcement Learning + Q-Learning
341	microkernel operating systems
342	”birthday party” ”nick cave”
343	fantasy novel goodkind book
344	XML database
345	Sex Pistols concert audience Manchester music scene
346	+unrealscript language api tutorial
347	+”state machine” figure moore mealy
348	drinking water abstraction +germany
349	proprietary implementation +protocol +wireless +security
350	Animal flight
351	”Chinese wedding” custom tradition
352	Faster-than-light travel
353	pseudocode for in-place sorting algorithm
354	novel adaptations for science fiction films
355	+”Best Actress” +”Academy Award” -Supporting -nominated winner film
356	use of natural language processing in information retrieval
357	babylonia babylonian assyriology
358	ontologies information retrieval semantic indexing
360	solar energy for domestic electricity and heating
361	”Europe after the second world war” + democracy
362	effect nuclear power plant accident

Table C.2: INEX CO topics with relevance assessments, Ad Hoc Track 2006, part 2

topic_id	title
363	Bob Dylan Eric Clapton
364	+mushroom poisonous poisoning
365	economy peru international investment tourism
366	Fourier transform applications
368	Hymenoptera +Apocrita -Symphyta +reproduction queen bees wasps hornets
369	Pillars of Hercules + Mythology
371	escaped convict ”William Buckley”
372	Purpose of voodoo rituals.
373	Australia’s involvement in Echelon spy network
374	Aid following the 2004 Tsunami.
375	states countries nuclear proliferation nonproliferation treaty npt
376	”diabetes mellitus” ”type 2” symptoms
378	+rules ”team sports” +indoor +ball world -football -basketball -handball -voleyball
379	”Helms-Burton law” ”United States” embargo against Cuba consequences economy
380	Symptoms: headache, fatigue, nausea
381	ubiquitous computing and application
382	”greek mythology” aphrodite
383	Informations about the city of Lyon in France
384	politics political albert einstein
385	arnold schwarzenegger stars cast
386	fencing +weapon
387	bridge types
388	rhinoplasty
390	Insomnia ”what are the causes” +sleep
391	Cricket ”How to Play”
392	Australian Aboriginals ”stolen generation”
395	September 11 ”conspiracy theories”
399	”mobile phone” country UMTS
400	”non violent” revolution country
401	movie award ”eddie murphy” ”jim carrey” ”robin williams”
402	country european capital
403	color television analog standard description
404	french france singer
405	”The Old Man and the Sea”
406	book architecture
407	”Football World Cup” +”Miracle of Bern”
409	Hybrid Vehicles -biology ”fuel efficiency” ”fuel sources” model engine
410	Routers and Switches +computer -travel -light network types history
411	+GSM, +CDMA, system,standard,clear battery coverage roaming price.
413	Coordinates and Population of capital cities of Europe

Table C.3: INEX CO topics with relevance assessments, Ad Hoc Track 2006, part 3

topic id	title	topic id	title
544	meaning of life	602	Webster's Dictionary
545	dance style	603	Tata Motors Company in India
546	19th century imperialism	607	law legislation act +nuclear -family
547	Greek revolution 1821	609	mechanism RAID storage
550	dna testing -forensic -maternity -paternity	610	Nikola Tesla inventions patents
551	pollen allergy	611	rotary engines in cars
552	keyboard instrument -electronic	613	wireless network security
553	spanish classical +guitar players	616	+Egypt museum pyramid
555	+Amsterdam picture image	617	+acne treatment side effects
556	vegetarian person -she -woman	624	"open source" information retrieval systems
557	electromagnetic waves	626	"Bayes Filter" +application
559	vodka producing countries	628	MBA school in Canada
561	Portuguese typical dishes	629	science fiction film
562	algerian war	634	vauban
563	Virginia Woolf novels	635	Linux Operating System
565	discovery "by chance" serendipity	636	Image File Formats
570	introduced animals	637	Java Programming Language
574	guitar tapping	641	Museum Picasso France
576	aircraft formation	642	social networks mining
577	genetically modified food safety	643	wikipedia vandalism
578	childbirth tradition	644	virtual museums
579	diet descriptions	646	records management" +metadata -system
580	"european basketball players" +nba	647	time travel theories
581	wine tasting	649	flower meaning
582	famous bouddhist places	650	ale
585	+"International brigades" Spanish Civil War	656	bilingualism children "language acquisition"
586	"magnetic levitation" technology	657	scrabble game rules
587	autistic spectrum disorder	659	technological singularity concept and implications
592	berbers of north africa	666	party primaries in the United States
595	car company	667	Wine regions in Europe
596	Jennifer Lopez	668	Codebreaking at Bletchley Park
597	expert on database	669	coin collecting
598	mahler symphony song	673	intrusion detection
600	Japanese culture food	675	Environmental Impacts of Earthquakes
601	Townships of Michigan	677	terracotta figures +horse

Table C.4: Assessed INEX CO topics, Ad Hoc Track 2008

topic id	title	phrasetitle
2009001	Nobel prize	”Nobel prize”
2009002	best movie	”best movie”
2009003	yoga exercise	”yoga exercise”
2009004	mean average precision reciprocal rank references precision recall proceedings journal	”mean average precision” ”reciprocal rank” ”precision recall” ”recall precision”
2009005	chemists physicists scientists alchemists periodic table elements	”periodic table”
2009006	opera singer italian spanish -soprano	”opera singer”
2009007	financial and social man made catastrophes adversity misfortune -”natural disaster”	-”natural disaster” ”financial misfortune” financial disaster” ”financial catastrophe” ”financial adversity” ”social disaster” ”social catastrophe”
2009008	israeli director actor actress film festival	”israeli director” ”israeli actor” ”israeli actress” ”film festival”
2009009	election +victory australian labor party state council -federal	”election victory” ”state election” ”council election” ”australian labor party”
2009010	applications bayesian networks bioinformatics	”bayesian networks”
2009011	olive oil health benefit	”olive oil” ”health benefit”
2009012	vitiligo pigment disorder cause treatment	”treatment of vitiligo”, ”cause of ”vitiligo” ”pigment disorder”
2009013	native american indian wars against colonial americans	”native american” ”american indian” ”wars against colonial americans”
2009014	content based image retrieval	”content based” ”image retrieval” ”content based image retrieval”
2009015	Voice over IP	none
2009016	cycle road skill race	”road bike” ”road race”
2009017	rent buy home	none
2009018	Dwyane Wade	”Dwyane Wade”
2009019	Latent semantic indexing	”latent semantic indexing”
2009020	IBM computer	”IBM computer”
2009021	wonder girls	”wonder girls”
2009022	Szechwan dish food cuisine	”Szechwan dish” ”Szechwan food” ”Szechwan cuisine”
2009023	”plays of Shakespeare”+Macbeth	”plays of Shakespeare”
2009024	cloud computing	”cloud computing”
2009025	scenic spot in Beijing	”scenic spot”
2009026	generalife gardens	none
2009027	Zhang Yimou	”Zhang Yimou”
2009028	fastest speed bike scooter car motorcycle	none
2009029	personality type career famous	”personality type”
2009030	popular dog cartoon character	”cartoon character”

Table C.5: INEX 2009 - Type A queries, part 1

topic id	title	phrasetitle
2009031	sabre	none
2009032	evidence theory dempster schafer	”evidence theory” ”dempster schafer”
2009033	Al-Andalus taifa kingdoms	”taifa kingdoms”
2009034	the evolution of the moon	none
2009035	Bermuda Triangle	”Bermuda Triangle”
2009036	notting hill film actors	”notting hill” ”film actors”
2009037	movies directed tarantino	”tarantino movie”
2009038	french colony africa independence	”french colony”
2009039	roman architecture	”roman architecture”
2009040	steam engine	”steam engine”
2009041	The Scythians	”the Scythians”
2009042	sun java	”sun java”
2009043	NASA missions	”NASA missions”
2009044	OpenGL Shading Language GLSL	”OpenGL Shading Language”
2009045	new age musician	”new age”
2009046	Penrose tiles tiling theory	”Penrose tiles” ”Tiling theory”
2009047	”Kali’s child” criticisms reviews Psychoanalysis of Ramakrishna’s mysticism	”Kali’s child” ”Psychoanalysis of Ramakrishna’s mysticism”
2009048	biometric technique	”biometric technique”
2009049	Chicago Symphony Orchestra	”Chicago Symphony Orchestra”
2009050	valentine’s day	”valentine’s day”
2009051	Rabindranath Tagore Bengali literature	”Rabindranath Tagore” ”Bengali literature”
2009052	newspaper spain headquarter Madrid	none
2009053	finland car industry manufacturer saab sisu	”car industry” ”car manufacturer”
2009054	tampere region tourist attractions	”tampere region” ”tourist attraction”
2009055	european union expansion	”european union”
2009056	higher education around the world	”higher education”
2009057	movie Slumdog Millionaire directed by Danny Boyle	”Slumdog Millionaire” ”Danny Boyle”
2009058	Tiananmen Square protest 1989	”Tiananmen Square” ”protest 1989”
2009059	failure tolerance in distributed systems	”failure tolerance” ”distributed systems”
2009060	hard disk technology	”hard disk”

Table C.6: INEX 2009 - Type A queries, part 2

topic id	title	phrasetitle
2009061	france second world war normandy	”second world war”
2009062	social network group selection	”group selection in social network” ”social network” ”group selection”
2009063	D-Day normandy invasion	”normandy invasion”
2009064	stock exhange insider trading crime	”stock exhange” ”insider trading”
2009065	sunflowers Vincent van Gogh	”Vincent van Gogh”
2009066	folk metal groups finland	”folk metal”
2009067	probabilistic models in information retrieval	”probabilistic models” ”information retrieval”
2009068	China great wall	”Great Wall”
2009069	Singer in Britain’s Got Talent	”Britain’s Got Talent”
2009070	health care reform plan	”health care reform” ”health care plan”
2009071	earthquake prediction	”earthquake prediction”
2009072	+professor ”information retrieval” ”computer science”	”information retrieval” ”computer science”
2009073	web link network analysis	”web link analysis” ”link analysis” ”network analysis”
2009074	web ranking scoring algorithm	”web ranking” ”scoring algorithm”
2009075	tourism in tunisia	none
2009076	sociology and social issues and aspects in science fiction	”social aspects” ”social issues” ”science fiction”
2009077	torrent client technology	”Torrent technology”
2009078	supervised machine learning algorithm	”supervised machine learning algorithm” ”machine learning”
2009079	dangerous paraben bisphenol-A	none
2009080	international game show formats	”game show” ”show formats”
2009081	Maya calendar	”Maya calendar”
2009082	south african nature reserve	”south african” ”nature reserve”
2009083	therapeutic food	”therapeutic food”
2009084	food allergy	”food allergy”
2009085	operating system +mutual +exclusion	”operating system” +”mutual exclusion”
2009086	airbus a380	none
2009087	history bordeaux	none
2009088	”hatha yoga” deity asana	”hatha yoga”
2009089	world wide web history	”world wide web”
2009090	Telephone history	none

Table C.7: INEX 2009 - Type A queries, part 3

topic id	title	phrasetitle
2009091	Himalaya trekking peak	none
2009092	ski +waxing -water -wave	”ski waxing”
2009093	French revolution	”French revolution”
2009094	global warming human activity	”global warming” ”human activity”
2009095	Weka software	none
2009096	Eiffel	none
2009097	location marcel duchamp work	”Marcel Duchamp”
2009098	Pandemic Death	none
2009099	movie houdini	none
2009100	search algorithm with plural keywords	”search algorithm” ”plural keywords”
2009101	alchemy in Asia including Japan China and India	”alchemy in Asia”
2009102	historical ninja stars	”ninja stars”
2009103	photograph world earliest	”earliest photograph”
2009104	lunar mare formation mechanism	”lunar mare” ”formation mechanism”
2009105	Musicians Jazz	”Jazz musicians”
2009106	+”amy macdonald” +love +song	”amy macdonald” ”love song”
2009107	design science sustainability renewable energy synergy	”design science” ”design science sustainability” ”renewable energy”
2009108	sustainability indicators metrics	”sustainability indicator” ”sustainability metric”
2009109	circus acts skills	”circus act” ”circus skills”
2009110	paul is dead hoax theory	+”paul is dead”
2009111	europe solar power facility	”solar power” ”facility in Europe”
2009112	rally car female OR woman driver	”rally car” ”female driver” ”woman driver”
2009113	Toy Story Buzz Lightyear 3D rendering Computer Generated Imagery	”Toy Story” ”Buzz Lightyear” ”3D rendering” ”Computer Generated Imagery”
2009114	self-portrait	”self portrait”
2009115	virtual museums	”virtual museum”

Table C.8: INEX 2009 - Type A queries, part 4

List of Figures

List of Tables

Bibliography

[AAS$^+$09] James Allan, Javed A. Aslam, Mark Sanderson, ChengXiang Zhai, and Justin Zobel, editors. *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*. ACM, 2009.

[AM06] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In Efthimiadis et al. [EDHJ06], pages 372–379.

[BAYS06] Chavdar Botev, Sihem Amer-Yahia, and Jayavel Shanmugasundaram. Expressiveness and performance of full-text search languages. In *EDBT*, pages 349–367, 2006.

[BBS10] Andreas Broschart, Klaus Berberich, and Ralf Schenkel. Evaluating the potential of explicit phrases for retrieval quality. In Cathal Gurrin, Yulan He, Gabriella Kazai, Udo Kruschwitz, Suzanne Little, Thomas Roelleke, Stefan M. Rüger, and Keith van Rijsbergen, editors, *ECIR*, volume 5993 of *Lecture Notes in Computer Science*, pages 623–626. Springer, 2010.

[BC05] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *CIKM*, pages 317–318. ACM, 2005.

[BC06] Stefan Büttcher and Charles L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management*, pages 182–189, 2006.

[BCH03a] Peter Bailey, Nick Craswell, and David Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Information Processing and Management*, 39(6):853–871, 2003.

[BCH$^+$03b] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434. ACM, 2003.

[BCL06] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In Efthimiadis et al. [EDHJ06], pages 621–622.

[BCS06] Stefan Büttcher, Charles L. A. Clarke, and Ian Soboroff. The TREC 2006 Terabyte Track. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*, volume Special Publication 500-272. National Institute of Standards and Technology (NIST), 2006.

[Bei07] Michel Beigbeder. ENSM-SE at INEX 2007: Scoring with proximity. In *Preproceedings of the 6th INEX Workshop*, pages 53–55, 2007.

[Bei10] Michel Beigbeder. Focused retrieval with proximity scoring. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 1755–1759. ACM, 2010.

[BFC10] Michael Bendersky, David Fisher, and W. Bruce Croft. UMass at TREC 2010 Web Track: Term dependence, spam filtering and quality bias. In *Web Track Notebook of the 19th Text REtrieval Conference*, 2010.

[BGM02] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE 2002*, pages 369–380, 2002.

[BMS$^+$06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 475–486. ACM, 2006.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[BRL06] Christopher J. C. Burges, Robert Ragno, and Quoc Viet Le. Learning to rank with nonsmooth cost functions. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 193–200. MIT Press, 2006.

[BS08a] Andreas Broschart and Ralf Schenkel. Effiziente Textsuche mit Positionsinformation. In Hagen Höpfner and Friederike Klan, editors, *Grundlagen von Datenbanken*, volume 01/2008 of *Technical Report*, pages 101–105. School of Information Technology, International University in Germany, 2008.

[BS08b] Andreas Broschart and Ralf Schenkel. Proximity-aware scoring for xml retrieval. In Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong, editors, *SIGIR*, pages 845–846. ACM, 2008.

[BS09] Andreas Broschart and Ralf Schenkel. Index tuning for efficient proximity-enhanced query processing. In Geva et al. [GKT10], pages 213–217.

[BS10] Andreas Broschart and Ralf Schenkel. MMCI at the TREC 2010 Web Track. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*. National Institute of Standards and Technology (NIST), 2010.

[BS11] Andreas Broschart and Ralf Schenkel. A novel hybrid index structure for efficient text retrieval. In Ma et al. [MNBY^{+}11], pages 1175–1176.

[BS12] Andreas Broschart and Ralf Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Transactions on Information Systems*, 30(1):5:1–5:32, 2012.

[BSM95] Chris Buckley, Amit Singhal, and Mandar Mitra. New retrieval approaches using SMART: TREC 4. In *TREC*, 1995.

[BST08] Andreas Broschart, Ralf Schenkel, and Martin Theobald. Experiments with proximity-aware scoring for XML retrieval at INEX 2008. In Geva et al. [GKT09], pages 29–32.

[BSTW07] Andreas Broschart, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. TopX @ INEX 2007. In Fuhr et al. [FKLT08], pages 49–56.

[BWZ02] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR*, pages 215–221. ACM, 2002.

[CCB95] James P. Callan, W. Bruce Croft, and John Broglio. TREC and Tipster experiments with Inquery. *Information Processing and Management*, 31(3):327–343, 1995.

[CCKS07] Surajit Chaudhuri, Kenneth Ward Church, Arnd Christian König, and Liying Sui. Heavy-tailed distributions and multi-keyword queries. In Kraaij et al. [KdVC^{+}07], pages 663–670.

[CCT97] Charles L. A. Clarke, Gordon V. Cormack, and Elizabeth A. Tudhope. Relevance ranking for one to three term queries. In *RIAO*, pages 388–401, 1997.

[CHKZ01] W. Bruce Croft, David J. Harper, Donald H. Kraft, and Justin Zobel, editors. *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*. ACM, 2001.

[CMS10] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines - Information Retrieval in Practice.* Addison Wesley, 2010.

[CO07] Ronan Cummins and Colm O'Riordan. An axiomatic study of learned weighting schemes. In *SIGIR Learning to Rank Workshop*, 2007.

[CO09] Ronan Cummins and Colm O'Riordan. Learning in a pairwise term-term proximity framework for information retrieval. In *SIGIR*, pages 251–258, 2009.

[CP06] Matthew Chang and Chung Keung Poon. Efficient phrase querying with common phrase index. In Mounia Lalmas, Andy MacFarlane, Stefan M. Rüger, Anastasios Tombros, Theodora Tsikrika, and Alexei Yavlinsky, editors, *ECIR*, volume 3936 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 2006.

[CSS05] Charles L. A. Clarke, Falk Scholer, and Ian Soboroff. The TREC 2005 Terabyte Track. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*, volume Special Publication 500-266. National Institute of Standards and Technology (NIST), 2005.

[CTL91] W. Bruce Croft, Howard R. Turtle, and David D. Lewis. The use of phrases and structured queries in information retrieval. In *SIGIR*, pages 32–45, 1991.

[CwH02] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 346–357. ACM, 2002.

[DG06a] Ludovic Denoyer and Patrick Gallinari. The Wikipedia XML Corpus. In Norbert Fuhr, Mounia Lalmas, and Andrew Trotman, editors, *INEX*, volume 4518 of *Lecture Notes in Computer Science*, pages 12–19. Springer, 2006.

[DG06b] Ludovic Denoyer and Patrick Gallinari. The Wikipedia XML Corpus. *SIGIR Forum*, 40(1):64–69, 2006.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[dKM99] Owen de Kretser and Alistair Moffat. Effective document presentation with a locality-based similarity heuristic. In *SIGIR*, pages 113–120. ACM, 1999.

[dKM04] Owen de Kretser and Alistair Moffat. Seft: a search engine for text. *Software - Practice and Experience*, 34(10):1011–1023, 2004.

[dMNZBY00] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[DS11] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In Ma et al. [MNBY$^+$11], pages 993–1002.

[EDHJ06] Efthimis N. Efthimiadis, Susan T. Dumais, David Hawking, and Kalervo Järvelin, editors. *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*. ACM, 2006.

[Fag87] Joel L. Fagan. Automatic phrase indexing for document retrieval: An examination of syntactic and non-syntactic methods. In *SIGIR*, pages 91–101, 1987.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[FKLT08] Norbert Fuhr, Jaap Kamps, Mounia Lalmas, and Andrew Trotman, editors. *Focused Access to XML Documents, 6th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2007, Dagstuhl Castle, Germany, December 17-19, 2007. Selected Papers*, volume 4862 of *Lecture Notes in Computer Science*. Springer, 2008.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 419–428. Morgan Kaufmann, 2000.

[GBK01] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628. IEEE Computer Society, 2001.

[GKT09] Shlomo Geva, Jaap Kamps, and Andrew Trotman, editors. *Advances in Focused Retrieval, 7th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2008, Dagstuhl Castle, Germany,* *December 15-18, 2008. Revised and Selected Papers*, volume 5631 of *Lecture Notes in Computer Science*. Springer, 2009.

[GKT10] Shlomo Geva, Jaap Kamps, and Andrew Trotman, editors. *Focused Retrieval and Evaluation, 8th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2009, Brisbane, Australia, December 7-9, 2009, Revised and Selected Papers*, volume 6203 of *Lecture Notes in Computer Science*. Springer, 2010.

[Haw00] David Hawking. Overview of the TREC-9 Web Track. In *TREC*, 2000.

[HBLH94] William Hersh, Chris Buckley, T. J. Leone, and David Hickam. Ohsumed: an interactive retrieval evaluation and new large test collection for research. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, pages 192–201, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[HCT98] David Hawking, Nick Craswell, and Paul B. Thistlewaite. Overview of trec-7 very large collection track. In *TREC*, pages 40–52, 1998.

[Hie98] Djoerd Hiemstra. A linguistically motivated probabilistic model of information retrieval. In Christos Nikolaou and Constantine Stephanidis, editors, *ECDL*, volume 1513 of *Lecture Notes in Computer Science*, pages 569–584. Springer, 1998.

[HVCB99] David Hawking, Ellen M. Voorhees, Nick Craswell, and Peter Bailey. Overview of the TREC-8 Web Track. In *TREC*, 1999.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58, 2008.

[JK02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[JM80] Frederick Jelinek and Robert L. Mercer. Interpolated estimation of markov source parameters from sparse data. *Pattern Recognition in Practice*, pages 381–397, 1980.

[KdVC$^+$07] Wessel Kraaij, Arjen P. de Vries, Charles L. A. Clarke, Norbert Fuhr, and Noriko Kando, editors. *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007*. ACM, 2007.

[KGT$^+$08] Jaap Kamps, Shlomo Geva, Andrew Trotman, Alan Woodley, and Marijn Koolen. Overview of the INEX 2008 Ad Hoc Track. In Geva et al. [GKT09], pages 1–28.

[KPK$^+$07] Jaap Kamps, Jovan Pehcevski, Gabriella Kazai, Mounia Lalmas, and Stephen Robertson. INEX 2007 evaluation measures. In Fuhr et al. [FKLT08], pages 24–33.

[KPSV09] Ravi Kumar, Kunal Punera, Torsten Suel, and Sergei Vassilvitskii. Top-aggregation using intersections of ranked inputs. In Ricardo A. Baeza-Yates, Paolo Boldi, Berthier A. Ribeiro-Neto, and Berkant Barla Cambazoglu, editors, *WSDM*, pages 222–231. ACM, 2009.

[Liu11] Tie-Yan Liu. *Learning to Rank for Information Retrieval.* Springer, Berlin Heidelberg, 2011.

[LLYM04] Shuang Liu, Fang Liu, Clement T. Yu, and Weiyi Meng. An effective approach to document retrieval via utilizing wordnet and recognizing phrases. In *SIGIR*, pages 266–272, 2004.

[LS05] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 257–266. ACM, 2005.

[LT07] Mounia Lalmas and Anastasios Tombros. INEX 2002 - 2006: Understanding XML retrieval evaluation. In Costantino Thanos, Francesca Borri, and Leonardo Candela, editors, *DELOS Conference*, volume 4877 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 2007.

[LZ01] John D. Lafferty and ChengXiang Zhai. Document language models, query models, and risk minimization for information retrieval. In Croft et al. [CHKZ01], pages 111–119.

[LZ09] Yuanhua Lv and ChengXiang Zhai. Positional language models for information retrieval. In Allan et al. [AAS$^+$09], pages 299–306.

[MBG04] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-queries over web-accessible databases. *ACM Transactions on Database Systems*, 29(2):319–362, 2004.

[MBSC97] Mandar Mitra, Chris Buckley, Amit Singhal, and Claire Cardie. An analysis of statistical and syntactic phrases. In *RIAO*, pages 200–217, 1997.

[MC05] Donald Metzler and W. Bruce Croft. A markov random field model for term dependencies. In Ricardo A. Baeza-Yates, Nivio Ziviani, Gary Marchionini, Alistair Moffat, and John Tait, editors, *SIGIR*, pages 472–479. ACM, 2005.

[MdR05] Gilad Mishne and Maarten de Rijke. Boosting web retrieval through query operations. In David E. Losada and Juan M. Fernández-Luna, editors, *ECIR*, volume 3408 of *Lecture Notes in Computer Science*, pages 502–516. Springer, 2005.

[Met06a] Donald Metzler. Estimation, sensitivity, and generalization in parameterized retrieval models. In Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu, editors, *CIKM*, pages 812–813. ACM, 2006.

[Met06b] Donald Metzler. Estimation, sensitivity, and generalization in parameterized retrieval models (extended version). *Technical report, University of Massachusetts*, 2006.

[MNBY$^+$11] Wei-Ying Ma, Jian-Yun Nie, Ricardo A. Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft, editors. *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*. ACM, 2011.

[Mon04] Christof Monz. Minimal span weighting retrieval for question answering. In Rob Gaizauskas, Mark Greenwood, and Mark Hepple, editors, *Proceedings of the SIGIR Workshop on Information Retrieval for Question Answering*, pages 23–30, 2004.

[MOT11] Craig MacDonald, Iadh Ounis, and Nicola Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Transactions on Information Systems*, 29(4):17:1–17:28, 2011.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[MSC06] Donald Metzler, Trevor Strohman, and W. Bruce Croft. Indri trec notebook 2006: Lessons learned from three terabyte tracks. In *TREC*, 2006.

[MSTC04] Donald Metzler, Trevor Strohman, Howard R. Turtle, and W. Bruce Croft. Indri at trec 2004: Terabyte track. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*, volume Special Publication 500-261. National Institute of Standards and Technology (NIST), 2004.

[NC10] Dong Nguyen and Jamie Callan. Combination of evidence for effective web search. In *Proceedings of the 19th Text REtrieval Conference*, 2010.

[PA97] R. Papka and J. Allan. Why bigger windows are better than small ones. Technical report, CIIR, 1997.

[PC98] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *SIGIR*, pages 275–281. ACM, 1998.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming.* lulu.com, 2008.

[PRL+07] Ivana Podnar, Martin Rajman, Toan Luu, Fabius Klemm, and Karl Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *ICDE*, pages 1096–1105. IEEE, 2007.

[RJ76] S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3):129–146, 1976.

[RS03] Yves Rasolofo and Jacques Savoy. Term proximity scoring for keyword-based retrieval systems. In Fabrizio Sebastiani, editor, *ECIR*, volume 2633 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 2003.

[RW94] Stephen E. Robertson and Steve Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In W. Bruce Croft and C. J. van Rijsbergen, editors, *SIGIR*, pages 232–241. ACM/Springer, 1994.

[RWHB+95] Stephen E. Robertson, Steve Walker, Micheline Hancock-Beaulieu, Mike Gatford, and A. Payne. Okapi at TREC-4. In *TREC*, 1995.

[RZT04] Stephen E. Robertson, Hugo Zaragoza, and Michael J. Taylor. Simple BM25 extension to multiple weighted fields. In *CIKM*, pages 42–49, 2004.

[SBH+07] Ralf Schenkel, Andreas Broschart, Seung Won Hwang, Martin Theobald, and Gerhard Weikum. Efficient text proximity search. In Nivio Ziviani and Ricardo A. Baeza-Yates, editors, *SPIRE*, volume 4726 of *Lecture Notes in Computer Science*, pages 287–299. Springer, 2007.

[SC07] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In Kraaij et al. [KdVC+07], pages 175–182.

[SCC+01] Aya Soffer, David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, and Yoëlle S. Maarek. Static index pruning for information retrieval systems. In Croft et al. [CHKZ01], pages 43–50.

[SI00] Satoshi Sekine and Hitoshi Isahara. IREX: IR and IE evaluation-based project in Japanese. In *The Second International Conference on Language Resources and Evaluation*, 2000.

[SKK10] Krysta Marie Svore, Pallika H. Kanani, and Nazan Khan. How good is a span of terms?: exploiting proximity to improve web retrieval. In Fabio Crestani, Stéphane Marchand-Maillet, Hsin-Hsi Chen, Efthimis N.

Efthimiadis, and Jacques Savoy, editors, *SIGIR*, pages 154–161. ACM, 2010.

[SSK07] Ralf Schenkel, Fabian M. Suchanek, and Gjergji Kasneci. Yawn: A semantically annotated wikipedia xml corpus. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 277–291. GI, 2007.

[SSLM$^+$09] Michal Shmueli-Scheuer, Chen Li, Yosi Mass, Haggai Roitman, Ralf Schenkel, and Gerhard Weikum. Best-effort top-k query processing under budgetary constraints. In *ICDE*, pages 928–939. IEEE, 2009.

[STW$^+$08] Ruihua Song, Michael J. Taylor, Ji-Rong Wen, Hsiao-Wuen Hon, and Yong Yu. Viewing term proximity from a different perspective. In Craig Macdonald, Iadh Ounis, Vassilis Plachouras, Ian Ruthven, and Ryen W. White, editors, *ECIR*, volume 4956 of *Lecture Notes in Computer Science*, pages 346–357. Springer, 2008.

[SWY75] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[TAS08] Martin Theobald, Mohammed AbuJarour, and Ralf Schenkel. TopX 2.0 at the INEX 2008 Efficiency Track. In Geva et al. [GKT09], pages 224–236.

[TAS09] Martin Theobald, Ablimit Aji, and Ralf Schenkel. TopX 2.0 at the INEX 2009 Ad-Hoc and Efficiency Tracks. In Geva et al. [GKT10], pages 218–228.

[TJG09] Andrew Trotman, Xiangfei Jia, and Shlomo Geva. Fast and effective focused retrieval. In Geva et al. [GKT10], pages 229–241.

[TMO10] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient dynamic pruning with proximity support. In *LSDS-IR Workshop*, pages 31–35, 2010.

[TSW05] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for TopX search. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 625–636. ACM, 2005.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 648–659. Morgan Kaufmann, 2004.

[TZ07] Tao Tao and ChengXiang Zhai. An exploration of proximity measures in information retrieval. In Kraaij et al. [KdVC$^+$07], pages 295–302.

[UIF$^+$08] Yukio Uematsu, Takafumi Inoue, Kengo Fujioka, Ryoji Kataoka, and Hayato Ohwada. Proximity scoring using sentence-based inverted index for practical full-text search. In Birte Christensen-Dalsgaard, Donatella Castelli, Bolette Ammitzbøll Jurik, and Joan Lippincott, editors, *ECDL*, volume 5173 of *Lecture Notes in Computer Science*, pages 308–319. Springer, 2008.

[VH97] Ellen M. Voorhees and Donna Harman. Overview of the Fifth Text REtrieval Conference (TREC-5). In *Proceedings of the 5th Text REtrieval Conference*, pages 1–28, 1997.

[VH00] Ellen M. Voorhees and Donna Harman. Overview of the Eighth Text REtrieval Conference (TREC-8), 2000.

[W$^+$99] Hugh E. Williams et al. What's next? index structures for efficient phrase querying. In *Australasian Database Conference*, pages 141–152, 1999.

[Was05] Larry Wasserman. *All of Statistics*. Springer, 2005.

[Whi09] Tom White. *Hadoop - The definite guide*. O'Reilly, 2009.

[WK09] Judith Winter and Gerold Kühne. Achieving high precisions with peer-to-peer is possible! In Geva et al. [GKT10], pages 242–253.

[WLM11] Lidan Wang, Jimmy J. Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In Ma et al. [MNBY$^+$11], pages 105–114.

[WMB99] I.H. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufman, San Francisco, 1999.

[WZB04] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, 22(4):573–594, 2004.

[XL07] Jun Xu and Hang Li. AdaRank: a boosting algorithm for information retrieval. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR 2007, pages 391–398, New York, NY, USA, 2007. ACM.

[YDS09a] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In Allan et al. [AAS$^+$09], pages 147–154.

[YDS09b] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 401–410, New York, NY, USA, 2009. ACM.

[YSZ$^+$10] Hao Yan, Shuming Shi, Fan Zhang, Torsten Suel, and Ji-Rong Wen. Efficient term proximity search with term-pair indexes. In Jimmy Huang, Nick Koudas, Gareth Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *CIKM*, pages 1229–1238. ACM, 2010.

[Z$^+$07] Wei Zhang et al. Recognition and classification of noun phrases in queries for effective retrieval. In *CIKM*, pages 711–720, 2007.

[ZL04] ChengXiang Zhai and John D. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems*, 22(2):179–214, 2004.

[ZM98] Justin Zobel and Alistair Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1):18–34, 1998.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56, 2006.

[ZSLW07] Mingjie Zhu, Shuming Shi, Mingjing Li, and Ji-Rong Wen. Effective top-k computation in retrieving structured documents with term-proximity support. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *CIKM*, pages 771–780. ACM, 2007.

[ZSYW08] Mingjie Zhu, Shuming Shi, Nenghai Yu, and Ji-Rong Wen. Can phrase indexing help to process non-phrase queries? In James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz, Key-Sun Choi, and Abdur Chowdhury, editors, *CIKM*, pages 679–688. ACM, 2008.

[ZY09] Jinglei Zhao and Yeogirl Yun. A proximity language model for information retrieval. In Allan et al. [AAS$^+$09], pages 291–298.