TopX

Efficient and Versatile Top-k Query Processing for Text, Structured, and Semistructured Data

Martin Theobald
Max-Planck-Institut für Informatik

Saarbrücken
April 2006

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Thorsten Herfet
Vorsitzender der Prüfungskommission	Prof. Dr. Christoph Koch
Erstgutachter	Prof. Dr. Gerhard Weikum
Zweitgutachter	Prof. Dr. Norbert Fuhr
Zweitgutachter	Prof. Dr. Michalis Vazirgiannis
Wissenschaftlicher Begleiter	Dr. Ralf Schenkel
Tag des Promotionskolloquiums	16. Mai 2006

Kurzfassung

TopX ist eine Top-k Suchmaschine für Text und XML Daten. Im Gegensatz zu Boole' schen Suchmaschinen terminiert TopX die Anfragebearbeitung, sobald die k besten Ergebnisobjekte im Hinblick auf eine mehrdimensionale Anfrage gefunden wurden. Die Hauptbeiträge dieser Arbeit teilen sich in vier Schwerpunkte basierend auf vorherigen Veröffentlichungen bei internationalen Konferenzen oder Workshops:

- Top-k Anfragebearbeitung mit probabilistischen Garantien.
- Zugriffsoptimierte Top-k Anfragebearbeitung.
- Dynamische und selbstoptimierende, inkrementelle Anfrageexpansion für Top-k Anfragebearbeitung.
- Effiziente Unterstützung für XML-Anfragen und Volltextsuche.

Unsere Experimente bestätigen die Vielseitigkeit und gesteigerte Effizienz unserer Verfahren gegenüber existierenden, führenden Ansätzen für eine weite Bandbreite von Anwendungen in der Informationssuche.

Abstract

TopX is a top-k retrieval engine for text and XML data. Unlike Boolean engines, it stops query processing as soon as it can safely determine the k top-ranked result objects according to a monotonous score aggregation function with respect to a multidimensional query. The main contributions of the thesis unfold into four main points, confirmed by previous publications at international conferences or workshops:

- Top-k query processing with probabilistic guarantees.
- Index-access optimized top-k query processing.
- Dynamic and self-tuning, incremental query expansion for top-k query processing.
- Efficient support for ranked XML retrieval and full-text search.

Our experiments demonstrate the viability and improved efficiency of our approach compared to existing related work for a broad variety of retrieval scenarios.

Zusammenfassung

Top-k Anfragen basierend auf dem Erstellen von nach Relevanz sortierten Ergebnislisten für mehrdimensionale Datensätze sind ein fundamentaler Baustein für viele Arten der Informationssuche, wobei die Anwendungen von Text und Datenintegration bis zur verteilten Aggregation von Netzwerkprotokollen und Sensordaten reichen. The Top-k-Anfrageszenarien, die in dieser Arbeit untersucht werden, arbeiten auf vorausberechneten, invertierten Indexlisten für die elementaren Bedingungen einer Anfrage und aggregieren die elementaren Relevanzwerte der Ergebniskandidaten zu einem Gesamtrelevanzwert. Eine der effizientesten und vielseitigsten Implementierungsmethoden in diesem Kontext ist Fagin's Familie der Schwellwertalgorithmen, die darauf zielen, die Indexzugriffe so früh wie möglich zu terminieren, basierend auf unteren und oberen Schranken für den schließlichen Gesamtwert eines Kandidaten. Im Gegensatz zu Boole'schen Auswertungsmodellen profitieren diese fortgeschrittenen Verfahren von einer nicht-konjunktiven Auswertungsstrategie, bei der ein Ergebniskandidat schwache Übereinstimmungen einiger Anfragebedingungen (einschließlich keiner Übereinstimmung) durch starke Treffer für andere Anfragebedingungen ausgleichen kann. Diese Arbeit präsentiert eine neuartige Suchmaschine, genannt *TopX*, für die effiziente Anfrageevaluation von Text, strukturierten und semistrukturierten (XML) Daten. Der Kern des TopX Anfragebearbeiters integriert und erweitert unterschiedliche, existierende Verfahren auf signifikante Art zur kostenbewussten Indexzugriffssteuerung in einer flexiblen Mehrprozessarchitektur.

Da das Ziel eines Benutzers beim Stellen einer Top-k-Anfrage typischerweise darin liegt, eine oder mehrere neue Informationseinheiten zu entdecken, besteht eine faszinierende Idee darin, approximative Top-k-Verfahren zu entwickeln, um die Ausführungskosten einer solchen Anfrage weiter zu senken. TopX stellt daher eine Reihe von approximativen Algorithmen basierend auf probabilistischen Argumenten vor und ermöglicht dadurch großartige Laufzeitgewinne mit einem geringen und kontrollierbaren Verlust an Ergebnispräzision. Beim Einlesen der Indexlisten des zugrunde liegenden Datenraumes in absteigender Reihenfolge der lokalen Ränge werden verschiedene Arten von Verteilungsfaltungen und daraus abgeleiteten Schranken untersucht, um vorherzusagen, wann es mit hoher Wahrscheinlichkeit sicher ist,

Ergebniskandidaten zu löschen und den Einlesevorgang frühzeitig abzubrechen. Unser Ansatz umfasst effizient berechenbare, geschlossene Formen von Verteilungsfaltungen für parametrisierte Verteilungen wie Poission, vielseitige probabilistische Garantien mit momentgenerierenden Funktionen und Chernoff-Hoeffding-Schranken, sowie flexible Histogramme, die dazu benutzt werden können, beliebige Verteilungen zu approximieren.

Die grundlegende Anfrageauswertung führt effiziente sequentielle Indexzugriffe aus, hat aber auch die Option, gezielte randomisierte Zugriffe für ausgewählte Kandidatenobjekte auszulösen, um deren schließlichen Gesamtrelevanzwert direkt zu ermitteln. Diese Auswertungsstrategie beinhaltet also die zielgesteuerte Planung von zwei Arten von Indexzugriffen, nämlich 1) die unterschiedliche Priorisierung der Indexlisten in den sequentiellen Zugriffen und 2) die Entscheidung, wann und für welche Kandidaten ein randomisierter Zugriff ausgelöst werden soll. In der bisher existierenden Literatur wurden diese beiden Aspekte nur einzeln und für spezialisierte Umgebungen untersucht. Diese Arbeit vermittelt eine integrierte Untersuchung dieser Zugriffsstrategien und entwickelt neue Methoden, die die bisherigen Verfahren deutlich verbessern. Unsere Hauptbeiträge in diesem Zusammenhang sind grundlegend neue Ansätze basierend auf einer dem Rucksack-Problem abgeleiteten Optimierung von sequentiellen Zugriffen und ein probabilistisches Kostenmodell für die Steuerung randomisierter Zugriffe. Unsere Verfahren können durch die Einbeziehung unseres probabilistischen Schätzers, unterschiedlicher Selektivitäten, sowie Korrelationen von Indexlisten weiter verfeinert werden.

Desweiteren präsentiert diese Arbeit ein neues Verfahren zur dynamischen und selbstoptimierenden Anfrageexpansion, das ebenfalls in unsere Top-k-Auswertungsstrategie eingebettet ist. Traditionelle Expansionstechniken wählen Expansionsterme deren thematische Ähnlichkeit zu dem ursprünglichen Term über einem bestimmten Schwellwert liegt und generieren dadurch eine disjunktive Anfrage von deutlich höherer Dimensionalität. Dies führt häufig zu den folgenden drei Problemen: 1) der Schwellwert muss manuell eingestellt werden, 2) die Gefahr, dass das ursprüngliche Thema der Anfrage durch zu aggressive Expansion zunehmend verwischt wird, und 3) die drastisch erhöhten Ausführungskosten einer hochdimensionalen Anfrage. Unsere Methoden adressieren diese drei Schwerpunkte, indem die invertierten Indexlisten zu einer Menge von Expansionstermen dynamisch und inkrementell Verschmolzen werden. Eine Prioritätsschlange wird zur effizienten Verwaltung der Kandidaten verwendet, und das frühzeitige Eliminieren von schwachen Kandidaten basiert entweder auf dem konservativen Schwellwertverfahren der ursprünglichen Top-k-Algorithmen oder wiederum auf probabilistischen Argumenten. Die vorgestellten Algorithmen werden eingebettet in zwei neuartige, spezialisierte Anfrageoperatoren.

Für die effiziente Erstellung von Ergebnisranglisten von XML Dokumenten über semistrukturierten aber nicht-schematischen Datensammlungen ist TopX in der Lage, die effiziente Form der Indexzugriffssteuerung mit einem Großteil an effizienten sequentiellen Zugriffen und nur einigen wenigen, kontrollierten randomisierten Zugriffen beizubehalten und um ein XML-spezifisches Kostenmodell zu erweitern, was eine einzigartige Charakteristik im Bereich der Top-k-Anfragebearbeitung für XML Daten ist. Die Schwierigkeiten, die bestehenden Top-k-Verfahren auf XML Daten zu übertragen bestehen darin, 1) Relevanzwerte für einzelne XML Elemente zu betrachten, diese aber auf der Dokumentebene zu aggregieren, 2) eine vage Interpretation der XML Inhalte mit strukturellen Bedingungen zu kombinieren, 3) einzelne Anfragebedingungen dynamisch zu lockern, falls zu wenige Ergebnisse alle Bedingungen erfüllen, sowie 4) die Anpassung der Selektivitätsschätzung sowohl für textuelle als auch strukturelle Inhalte und deren Einfluss auf die Auswertungsstrategien. TopX adressiert diese Anforderungen durch die gezielte Vorausberechnung von elementaren Relevanzwerten und Pfadinformationen in einer spezialisierten Indexstruktur, durch die weitgehende Vermeidung oder Hinauszögerung der Auswertung von teuren Pfadbedingungen, um das sequentielle Zugriffsmuster auf die Indexlisten beizubehalten, und durch das selektive Ausführen von randomisierten Zugriffen zu einem kostengünstigen Zeitpunkt.

Ausführliche Experimente mit verschiedenen, realistischen Datensammlungen sowie offiziellen Benchmarks wie TREC oder INEX sowohl für Text als auch semistrukturierte Daten bestätigen die verbesserte Effizienz, Effektivität und Skalierbarkeit unserer Verfahren.

Summary

Top-k queries based on ranking elements of multidimensional datasets are a fundamental building block for many kinds of information discovery, with applications ranging from text and data integration to distributed aggregation of network logs and sensor data. The top-k retrieval scenarios we investigate operate on precomputed, inverted index lists for a query's elementary conditions and aggregate scores for result candidates. One of the most efficient and versatile methods in this setting is Fagin's family of threshold algorithms (TA), which aim to terminate the index scans as early as possible based on lower and upper bounds for the final scores of result candidates. As opposed to Boolean retrieval models, these advanced retrieval techniques greatly benefit from a non-conjunctive evaluation strategy, where a result object can compensate weak matches for some query conditions (including 0 scores) through high scores at other query conditions. This thesis presents a novel engine, coined *TopX*, for efficient, ranked retrieval of text, structured, and semistructured data (XML). The TopX core query processor seamlessly integrates existing approaches for cost-aware index access scheduling and substantially extends these in a multithreaded architecture.

Since the user's goal behind top-k queries is to identify one or a few relevant and novel data items, it is intriguing to use *approximate variants of TA* to reduce runtime costs. TopX introduces a family of approximate top-k algorithms based on probabilistic arguments, thus greatly speeding up queries with a small and controllable loss in retrieval precision. When scanning index lists of the underlying multidimensional data space in descending order of local scores, various forms of convolutions and derived bounds are employed to predict when it is safe, with high probability, to drop candidate items and to prune the index scans. Our approach investigates efficiently evaluable closed-form convolutions for parameterized score distributions such as Poisson, versatile probabilistic guarantees employing moment-generating functions and Chernoff-Hoeffding bounds, as well as flexible histograms that can be employed to approximate arbitrary distributions.

The basic query processing performs sequential disk accesses for sorted index scans, but also has the option of performing random accesses for selected data objects to resolve score uncertainty. This entails *scheduling for*

the two kinds of accesses, namely 1) the prioritization of different index lists in the sequential accesses, and 2) the decision on when to perform random accesses and for which candidate objects. The prior literature has studied some of these scheduling issues, but only for each of the two access types in isolation. The thesis takes an integrated view of these scheduling issues and develops novel strategies that outperform prior proposals by a large margin. Our main contributions are new, principled scheduling methods based on a Knapsack-related optimization for sequential accesses and a probabilistic cost model for random accesses. The methods can be further boosted by harnessing our probabilistic estimators for scores, different selectivities, and index list correlations.

We present a novel approach for *dynamic and self-tuning query expansion* that is natively embedded into our top-k query processor with early candidate pruning. Traditional query expansion methods select expansion terms whose thematic similarity to the original query terms is above some specified threshold, thus generating a disjunctive query with much higher dimensionality. This poses three major problems: 1) the need for hand-tuning the expansion threshold, 2) the potential topic dilution through overly aggressive expansion, and 3) the drastically increased execution cost of a high-dimensional query. The methods developed here addresses all three problems by dynamically and incrementally merging the inverted lists for the potential expansion terms with the lists for the original query terms. A priority queue is used for maintaining result candidates, and the pruning of candidates is either based on the conservative threshold condition of the TA-style algorithms or on probabilistic arguments. The proposed algorithms are encapsulated into two novel types of pipelined and non-blocking query operators, namely the Incremental Merge and nested top-k operators.

For *efficient ranked retrieval of XML documents* over semistructured but non-schematic data collections, TopX is able to retain the paradigm of threshold algorithms for top-k query processing, with a focus on inexpensive sequential accesses to index lists and only a few judiciously scheduled random accesses which is a unique characteristic among the XML engines we are aware of. The difficulties in applying the existing top-k algorithms to XML data lie in 1) the need to consider scores for XML elements while aggregating them at the document level, 2) the combination of vague content conditions with XML path conditions, 3) the need to relax query conditions if too few results satisfy all conditions, and 4) the selectivity estimation for both content and structure conditions and their impact on the evaluation strategies. TopX addresses these issues by precomputing score and path information in an appropriately designed index structure, by largely avoiding or postponing the evaluation of expensive path conditions so as to preserve the sequential

access pattern on index lists, and by selectively scheduling random accesses when they are cost-beneficial.

Extensive experiments on various real-world data collections, as well as official benchmark settings such as the TREC and INEX benchmarks for both text and semistructured data demonstrate the increased efficiency, effectiveness, and scalability of our approach.

Contents

1 Introduction **13**
1.1 Top-k Applications . . . 13
1.2 Classification of Top-k Algorithms . . . 14
1.2.1 Sequential vs. Random Access . . . 14
1.2.2 Top-k Selection vs. Top-k Join Queries . . . 15
1.3 TopX System Overview . . . 15
1.3.1 TopX Components . . . 16
1.3.2 Basic Top-k Query Processing . . . 19
1.4 Related Work . . . 22
1.4.1 Top-k Query Processing . . . 22
1.4.2 Approximative Top-k and Efficient IR . . . 27
1.4.3 Rank-Join Optimization . . . 29
1.4.4 Query Expansion . . . 31
1.4.5 XML IR . . . 32
1.5 Contributions . . . 35
1.5.1 Top-k Query Processing with Probabilistic Guarantees 35
1.5.2 Index Access Scheduling . . . 35
1.5.3 Dynamic & Self-tuning Incremental Query Expansions 36
1.5.4 Efficient XML Full-Text Search . . . 37
1.5.5 Selected Publications . . . 39
1.6 Overview of the Thesis . . . 40

2 Data & Query Model **41**
2.1 Data Model . . . 41
2.1.1 Text . . . 42
2.1.2 Structured Data . . . 44
2.1.3 Extensible Markup Language (XML) 1.0 . . . 44
2.1.4 Full-Content Text Model . . . 48
2.2 Query Language . . . 50
2.2.1 XPath 1.0 . . . 50
2.2.2 XPath 2.0 – Full-Text Extension . . . 55

2.2.3 Narrowed Extended XPath I (NEXI) 55
2.3 Relational Schemata for Text and Semistructured Data 58
2.3.1 Text Schema 58
2.3.2 Structural Indexes for XML 58
2.3.3 Combined Inverted Block-Index for XML 62

3 Relevance Scoring Model 66
3.1 Vector Space Model 68
3.1.1 TF·IDF Family of Scoring Functions 69
3.1.2 Vector Space Aggregations 71
3.2 Probabilistic Scoring Models 73
3.2.1 Probabilistic IR 73
3.2.2 Okapi BM25 76
3.3 Combined Scoring Models – Web IR 78
3.3.1 Multiple Weighted Fields 78
3.3.2 Link Structure & Anchor Texts 79
3.3.3 Global Document Weights 80
3.4 Scoring Models for Semistructured Data 83
3.4.1 Content Scores 83
3.4.2 Structural Scores 86
3.4.3 Common Framework 87
3.5 Query Term Weights & Boosting Factors 88
3.5.1 Relevance Feedback 89
3.5.2 Negative Query Weights 90

4 TopX Core Query Processor 91
4.1 Conjunctive vs. Andish Query Evaluations 92
4.1.1 Mixed Mandatory & Optional Query Conditions 92
4.1.2 Adaptive Min-k Thresholds 93
4.2 Expensive Predicates 94
4.2.1 Random Access Scheduling for Expensive Predicates . 94
4.2.2 Negation 96
4.2.3 Phrase Matching 97
4.2.4 Frequent Terms 98
4.3 Multi-threaded Top-k Query Processing 99
4.3.1 Main Thread 102
4.3.2 Scan Threads 104
4.3.3 Buffer Threads 105
4.3.4 Threshold & Continuous Queries 106

5 Probabilistic Candidate Pruning **111**
5.1 Top-k Query Processing with Probabilistic Guarantees 112
5.1.1 Convolutions 114
5.2 Predictors for Aggregated Scores 117
5.2.1 Chernoff-Hoeffding Bounds for Uniform Distributions . 117
5.2.2 Poisson Estimators 120
5.2.3 Histograms 121
5.2.4 Extensions and Generalizations 124
5.3 Efficient Queue Management 126
5.3.1 Conservative Algorithm 127
5.3.2 Aggressive Algorithm 128
5.3.3 Progressive Algorithm 129
5.3.4 Common Framework 130
5.4 Top-k Guarantees for Probabilistic Candidate Pruning 132

6 Index Access Scheduling **135**
6.1 Index-Optimized Top-k Query Processing 136
6.1.1 Adaptive Index Access Scheduling 136
6.1.2 Extended Classification of Threshold Algorithms 137
6.2 Probabilistic Extensions 139
6.2.1 Selectivity Estimator 139
6.2.2 Combined Score Predictor & Selectivity Estimator . . . 140
6.2.3 Feature Correlations 141
6.3 Sorted Access Scheduling 142
6.3.1 Knapsack Scheduling for Score Reduction 143
6.3.2 Knapsack Scheduling for Benefit Aggregation 145
6.4 Random Access Scheduling 147
6.4.1 Last-Probing 147
6.4.2 Ben-Probing 150

7 Dynamic & Self-tuning Query Expansion **154**
7.1 Static vs. Dynamic Query Expansion 155
7.1.1 Static Expansions & Topic Drifts 155
7.1.2 Dynamic & Incremental Query Expansion 157
7.2 Thesaurus-based Query Expansion 158
7.2.1 Word Sense Disambiguation 159
7.2.2 Similarity Joins 163
7.3 Unified Ontology Service 165
7.4 Incremental Merge Operator 166
7.4.1 Max-Score Aggregation 167
7.4.2 Incremental Merge Algorithm 168

7.4.3 Sorted Access for Dynamic Expansions 169
7.4.4 Random Access for Dynamic Expansions 170
7.5 Nested Top-k Operator 171
7.5.1 Dynamic Index Lists 172
7.5.2 Phrase Matching 173
7.5.3 Nested Top-k Algorithm 174
7.6 Probabilistic Extensions 178
7.6.1 Selectivity Estimator for Incremental Merge 179
7.6.2 Meta Histograms for Incremental Merge 180

8 Top-k Query Processing for XML 183
8.1 Challenges in Efficient XML IR 184
8.1.1 An XML IR Example Scenario 184
8.1.2 Requirements & Solutions Overview 186
8.1.3 Boolean XPath vs. XML IR 190
8.2 Query Decomposition & Index Block-Scans 191
8.2.1 Query Decomposition & Rewriting 191
8.2.2 Schema Mapping & Index Structures 194
8.2.3 Sorted Access for Element Blocks 196
8.2.4 Random Access for Element Blocks 198
8.3 Structure-aware Top-k Query Processing 199
8.3.1 TopX Query Processing by Example 200
8.3.2 In-Memory Structural Joins 202
8.3.3 Incremental Path Tests 204
8.3.4 Virtual Navigational Elements 209
8.3.5 Complexity 211
8.3.6 Element Retrieval 213
8.4 Random Access Scheduling for Structural Conditions 214
8.4.1 Min-Probing 215
8.4.2 Ben-Probing 216
8.5 Dynamic Query Expansion for Content & Structure 222
8.5.1 Incremental Merge & Structural Joins 222
8.5.2 Hybrid Index Structures 225

9 Experimental Evaluation 230
9.1 Hardware & Software Setup 230
9.2 TREC - Text REtrieval Conference 230
9.2.1 TREC Data Collections 231
9.2.2 Topic Format 233
9.3 INEX – INitiative for the Evaluation of XML Retrieval 234
9.3.1 INEX Collection 234

9.4 Highly Structured Collections 235
9.4.1 IMDB Collection – Relational 235
9.4.2 IMDB Collection – Semistructured 236
9.4.3 WorldCup HTTP Logs – Relational 236
9.5 Collections Summary 236
9.5.1 INEX Evaluation Strategies 237
9.6 Evaluation Metrics 240
9.7 Text IR 243
9.7.1 Probabilistic Candidate Pruning 243
9.7.2 Index Access Scheduling 253
9.7.3 Query Expansion 261
9.7.4 TREC 2004 268
9.7.5 TREC 2005 273
9.8 XML IR 274
9.8.1 Setup & Competitors 274
9.8.2 Strict Content & Structure Queries 279
9.8.3 Strict Content & Structure with Probabilistic Pruning 283
9.8.4 INEX 2005 285

10 Conclusions 290
10.1 Open Issues & Future Work 291
10.2 Concluding Remarks 293

A APPENDIX 294
A.1 Database Tables & Index Structures (DDL) 294
A.1.1 Text Schema 294
A.1.2 XML Schema 294
A.2 OpenMaple Scripts for Chernoff-Hoeffding Bounds 295
A.2.1 Chernoff-Hoeffding Bounds 295
A.2.2 Generalized Chernoff-Hoeffding Bounds 295
A.3 Index Access Scheduling 296
A.3.1 NP-hardness of the Sorted-Access Scheduling Problem 296
A.3.2 Lower Bound for the Index Access Scheduling Problem 298
A.4 Customized Queries 300
A.4.1 IMDB Relational Queries 300
A.4.2 IMDB NEXI Queries 301
A.4.3 Extended GOV (XGOV) Queries 302

Chapter 1

Introduction

1.1 Top-k Applications

Top-k queries based on ranking elements of multidimensional datasets are a fundamental building block for many kinds of information discovery. The best-known general-purpose algorithm for evaluating top-k queries is Fagin's threshold algorithm (TA) [FLN01], which has been independently proposed also by Nepal et al. [NR99] and Güntzer et al. [GBK00], whereas early, approximative variants of these algorithms have been proposed by Buckley [BL85] and Pfeifer et al. [PF95].

Top-k queries on multidimensional datasets compute the k most relevant or interesting results to a partial-match query, based on similarity scores of attribute values with regard to elementary query conditions and a monotonic score aggregation function such as weighted summation. This fundamental building block for information discovery arises in many important application classes such as

1) Web, intranet, or desktop search with scores based on word-occurrence statistics and possibly combining criteria like text-based relevance, link-based authority, and recency,

2) multimedia similarity search on high-dimensional feature vectors of images, music, or video, or

3) preference queries over structured and semistructured data such as product catalogs or customer support data (the latter having a major text component as well).

TA assumes that each attribute of the multidimensional data space has a precomputed index list by which one can access the data items in descending

order of the "local" score for the given attribute with regard to an elementary query condition, for example, the TF·IDF-based score for a text keyword condition "Trumpet", a thesaurus-based or feedback-driven similarity for categorical attribute conditions such as $Genre = Jazz$, or the absolute distance for numerical attribute conditions such as $Year = 1970$. In contrast to classic database management systems (DBMS), that first join all input tuples and then sort the output according to some aggregation function, top-k query processors aim at optimizing query executions when only a small subset of the top-scored result objects with respect to the aggregated scores (referred to as top-k) is required. Typically, these algorithms maintain and continuously update a set of the intermediate top-ranked results during query processing and try to derive a score *threshold* to determine when it is safe to terminate index scans and return the best objects based on what has been seen "so far", i.e., without having to completely scan all input sources, thus significantly reducing query execution costs and runtimes.

1.2 Classification of Top-k Algorithms

1.2.1 Sequential vs. Random Access

In general, the top-k or so-called *rank aggregation* algorithms proposed in the literature can be classified according to two orthogonal criteria. The first classification is based on the type of access available on the input lists. Each ranked input can support sequential and/or random access. Sequential access enables object retrieval in a descending order of their scores and is typically required to provide at least one ranked input source that helps to derive an upper score bound (and, hence, a threshold) for all candidate objects yet unseen in the sequential scans. Therefore, this mode is also often referred to as *sorted access*.

Random access enables probing or querying an input to retrieve a score of a given object directly. For example, the No-Random-Access (NRA) algorithm, originally introduced by Fagin [Fag99] and refined in [FLN01, FLN03], assumes only sorted access on the ranked inputs, while the TA algorithm, also initially introduced in [Fag99], assumes the availability of both random access and sorted access on all inputs. On the other hand, the algorithms introduced by Bruno et al. [BCG02, MBG04] and Chang and Hwang [CwH02] assume that at least one source has sorted access capability while other sources may have only random access (probing) available.

1.2.2 Top-k Selection vs. Top-k Join Queries

The second classification of rank aggregation algorithms is based on the assumptions on the underlying ranked objects. In the first category, all input sources share information about the same set of objects ranked according to different criteria. Hence, all the inputs can be viewed as one list of objects, where each object has a set of score attributes. The output is the same set of objects ranked on a combination (aggregation) of these score attributes. We refer to this problem as *top-k selection*. Most of the proposed algorithms belong to this category, e.g., Fagin's work [FLN01, FLN03], Nepal and Ramakrishna [NR99], and Güntzer et al. [GBK00].

In the second category of algorithms, e.g., Natsev et al. [NCS+01] and Ilyas et al. [IAE03, LCIS05], each input source potentially contains a different set of objects. A "join" condition among objects in different inputs joins them into one output join result. Each join result has a combined score that is computed from the scores of the participating objects. The goal is to produce the top-k join results. We refer to this problem as *top-k join*.

1.3 TopX System Overview

Following the aforementioned classification of top-k algorithms, this thesis investigates a combined sorted and random access model for top-k selection queries, thus following and substantially extending Fagin's predominant work on threshold algorithms. Our efforts led to a prototype system which we coined *TopX*.

With its seamless integration of efficient query evaluation and versatile scoring models for ranked result output, TopX resides at the very synapse of database (DB) engineering and information retrieval (IR). As for the DB viewpoint, we aim at providing an efficient algorithmic basis for scalable, top-k-style processing of large amounts of data. Our focus therein lies on adaptive, disk-oriented cost models for accessing large, disk-resident index structures, with highly developed solutions for storing and efficiently querying large document collections (possibly in the order of Terabytes). We leverage the observation that sequential disk I/O largely benefits from asynchronous prefetching and high locality in the hardware's and processor's cache hierarchy; so it has much lower amortized costs than random disk access that is inevitably requiring additional index structures and key lookups for individual object identifiers. Thus, our query processing methods focus on these inexpensive sequential disk IO, but with a controlled amount of random accesses that are crucial for resolving the final score of particularly promising

candidate objects and to resolve unclarity among the final result ranks.

As for the IR point-of-view, TopX provides a complete framework for indexing and efficiently searching text, semistructured, and structured data. The TopX query processor comes with support for mixed conjunctive and non-conjunctive, IR-style retrieval options, as well as expensive text predicates such as phrases, mandatory terms, or term and phrase negations. It is a self-contained query engine with support for advanced IR techniques and various scoring models for all sorts of IR applications. It provides a whole bunch of partly novel, effective scoring approaches for Web IR, structured attributes, and ranked XML retrieval including XML full-text search.

Our approach assumes that all values for all individual attributes are *fully precomputed* and stored on disk in an appropriate schema, either in a relational database system or in a more object-oriented way using inverted files, including optional index list meta data such as score histograms, correlation statistics, and quantified attribute similarities, e.g., a large thesaurus capturing term relations and their similarities. Then, at query processing time, score aggregations and result ranking for *multi-attribute queries* takes place on top of the database systems and is exclusively part of the TopX engine. Note that top-k query processing for single-dimensional queries simply resolves in fetching the first k entries from the respective inverted list in this setting.

The TopX framework comprises a full-fledged solution for desktop, intranet, and Web search. It is a comprehensive framework for indexing and querying large data collections and has been extensively studied in our experiments on various real-world data collections, including our participation at the two major benchmark series in IR, namely the Text REtrieval Conference (TREC) on text IR and the Initiative for the Evaluation of XML Retrieval (INEX) focusing on XML IR. It comes with different, general-purpose Web and file crawlers for indexing text and XML data in a generalized relational schema and further includes specialized indexers for the various benchmark-specific TREC and INEX collections with collection-specific metadata extraction.

1.3.1 TopX Components

Figure 1.1 depicts the TopX main components. The TopX core query processor is in charge of the bookkeeping of intermediate results and coordinates the sequential and random index lists accesses in a multi-threaded architecture. It provides the algorithmic basis for exact and efficient top-k query evaluations with early threshold termination, with the option of gradually plugging in specialized components:

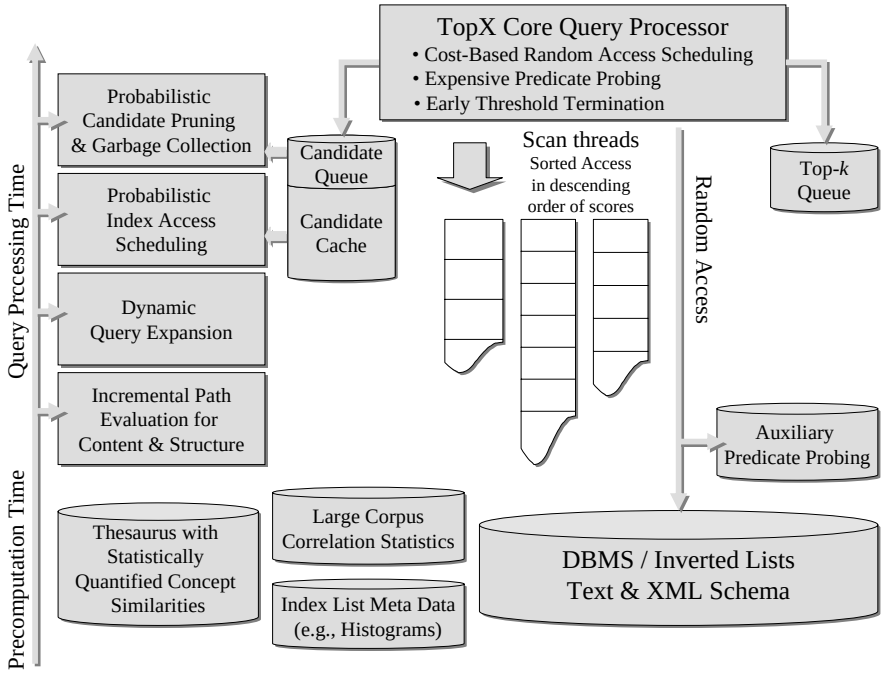


Figure 1.1: TopX components.

- Probabilistic score predictors for early candidate pruning.
- Probabilistically derived sorted and random access scheduling decisions.
- Dynamic query expansion using specialized query operators.
- Efficient support for XML full-text search.

When available, different probabilistic extensions can be incorporated for the probabilistic pruning components and the cost estimators for index access scheduling:

- Basic selectivity estimators.
- Index list histograms or parameterized score estimators with convolutions for aggregated scores.
- Index list correlations.
- Structural selectivities for basic structural XML patterns.

In the simplest case, the index list metadata merely consist of the information about the list length, i.e., the term selectivity or so-called document frequency (DF) in IR. Then we can compare this information with the current scan positions in the inverted lists to derive an initial probability of seeing a particular candidate in one or more of these lists. With the presence of fine-grained score histograms or a parameterized score estimator such as a compact Poisson-estimator, we can also estimate how many items will be able to exceed some required score threshold. With explicit information about feature correlations, e.g., in the form of term co-occurrence statistics, we can even refine these predictions and estimate the probability that a candidate object might exceed a certain score threshold given that it has been seen in one or more (potentially highly correlated) lists already, and so on. These extensions can be plugged in gradually and independently, e.g., using histograms only or selectivities only, thus yielding a powerful query engine for text, structured, and semistructured data.

Probabilistic Candidate Pruning

Since the user's goal behind top-k queries is to identify one or a few (hence, k) relevant and novel data items, it is intriguing to allow IR-style approximate variants of the threshold family of algorithms to reduce runtime costs. The thesis introduces a family of approximate top-k algorithms based on probabilistic arguments. When scanning index lists of the underlying multidimensional data space in descending order of local scores, various forms of convolutions and derived bounds are employed to predict when it is safe, with high probability, to drop candidate items and to prune the index scans for early algorithm termination. We show that these probabilistic candidate pruning techniques provide up to two orders of magnitude performance gains with a controllable loss in result quality and a very good quality/runtime ratio.

Index Access Scheduling

The basic top-k query processor performs sequential disk accesses for sorted index scans, but also has the option of performing random accesses to directly resolve score uncertainty of data objects. This entails scheduling for the two kinds of accesses: 1) the prioritization of different index lists in the sequential accesses, and 2) the decision on when to perform random accesses and for which candidates. Both types involve highly specialized probabilistic cost models, thus leading to individual index access scheduling decisions for result candidates and – as opposed to the probabilistic pruning component – can

substantially improve the performance of the retrieval engine with no loss in result quality.

Dynamic Query Expansion

For numerical or categorical attribute-value conditions that are not perfectly matched, the query processor could consider "alternative" values in ascending order of some notion of similarity to the original value of the query. For example, when searching for $Year = 1999$, after exhausting the index list for the value 1999, the next best lists are those for 1998, 2000, 1997, 2001, and so on. Although this relaxation involves additional index lists capturing the alternative value, we can treat this procedure as if it were a single index scan (for one of the m query dimensions), where the list for 1999 is conceptually extended by "neighboring" lists. In a static precomputation, this is easily done by adhering entries to the original index list. For truly *dynamic expansions* of text terms or categorical values, more sophisticated incremental merging techniques for these inverted lists are required that dynamically create the output list for such an expansion.

XML Full-Text Search

All the previous work finally finds its way into a structure-aware query processor for XML data and IR-style full-text search. Our XML-specific solutions comprise basic extensions for the storage of semistructured data in a relational schema and a true top-k query processor for the retrieval at both document and element granularity. Since the algorithmic paradigm we pursue focuses on inexpensive sequential disk I/O, our algorithms have to efficiently deal with uncertainty in both the structural and content-related conditions of multidimensional, hierarchical path queries. We thus present a novel approach for incremental path evaluations in a full-fledged path engine with support for all XPath axes. Our XML-specific solutions include the support for probabilistic candidate pruning and extensions for the structure-aware random access scheduling for XML data.

1.3.2 Basic Top-k Query Processing

In order to find the top-k matches for multidimensional queries, scoring, and ranking them, TopX scans all relevant index lists in an interleaved manner. In each scan step, when the engine sees the score for a data item in one list, it is hash-joined with the partial scores for the same data item previously seen in other index lists and aggregated into a *global score*. The basic query

processing model is based on the NRA and CA variants of the TA family of algorithms (see Section 1.4.1). Note that the way we focus on inexpensive sequential scans leaves *uncertainty* about the final scores of candidates and therefore implies some form of bookkeeping or queuing not only for the intermediate top-k results, but for all candidates that may still qualify for the final top-k.

To retrieve the top-k ranked results of an m-dimensional query, TA scans all query-relevant input lists L_i in an interleaved manner. Without loss of generality, we assume that these are the index lists numbered L_1 through L_m. When scanning the m index lists, the query processor collects candidates for the query result and maintains them in two priority queues, one for the *current top-k items* and another one for *all other candidates* that could still make it into the final top-k. For simpler presentation, we assume that the score aggregation function is simple summation (but it is easy to extend this to other monotonic functions). Then the query processor maintains the following state information:

- the score values $high_i$ at the current cursor positions, which serve as upper bounds for the unknown scores in the lists' tails,
- a set of current top-k items, d_1 through d_k (renumbered to reflect their current ranks) and a set of data items d_j for $j = k+1..k+q$ in the current candidate queue Q, following a basic data structure containing
  - a set of evaluated query dimensions (or index lists) $E(d)$ in which d has already been seen during the sequential scans or by random lookups,
  - a set of remainder query dimensions $\bar{E}(d)$ for which the score of d is still unknown,
  - a lower bound $worstscore(d)$ for the total score of d which is the sum of the scores $s_i(d)$ for $i \in E(d)$,

$$worstscore(d) \quad := \quad \sum_{i \in E(d)} s_i(d) \tag{1.1}$$

  - an upper bound $bestscore(d)$ for the total score of d which is equal to

$$bestscore(d) \quad := \quad worstscore(d) + \sum_{\nu \in \bar{E}(d)} high_\nu \tag{1.2}$$

    (and not actually stored but rather computed from $worstscore(d)$ and the current $high_\nu$ values whenever needed).

In addition, the following information is derived at each step:

- the minimum worstscore *min-k* of the current top-k docs, which serves as the stopping threshold,
- and for each candidate, a score deficit $\delta(d) = min\text{-}k - worstscore(d)$ that d would have to reach in order to qualify for the current top-k.

The invariant that separates the top-k list from the remaining candidates is that the rank-k worstscore of the top-k queue is at least as high as the best worstscore in the candidate queue. The algorithm can safely terminate, yielding the correct top-k results, when the maximum bestscore of the candidate queue is not larger than the rank-k worstscore of the current top-k, i.e., when

$$\min_{d \in top\text{-}k} \{worstscore(d)\} =: min\text{-}k \geq \max_{c \in Q} \{bestscore(c)\} \tag{1.3}$$

We will refer to Equation 1.3 as the *min-k threshold test*. More generally, whenever a candidate in the queue Q has a bestscore that is not higher than *min-k*, this candidate can be pruned from the queue. Early termination (i.e., the point when the queue becomes empty) is one goal of efficient top-k processing, but early pruning to keep the queue and its memory consumption small is an equally important goal (and is not necessarily implied by early termination). Figure 1.2 shows the corresponding bookkeeping for the intermediate top-k result queue and the candidate queue.

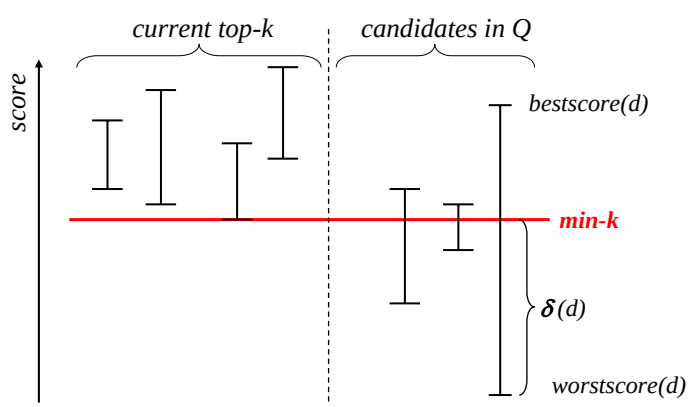


Figure 1.2: Top-k and candidate bookkeeping.

Conceptually, we maintain two priority queues in-memory to implement the threshold test: one for the current top-k results with items prioritized in ascending order of worstscores, and one for the currently best candidates with

items prioritized in descending order of bestscores. The first queue contains only items whose $worstscore(d) \geq min\text{-}k$ and the latter has items whose $worstscore(d) \leq mink$ but whose $bestscore(d) > min\text{-}k$. Then testing the top-prioritized items from the two queues as denoted by the above threshold test equation yields a safe stopping condition at any time of the query processing. Ties among scores may be broken by using the concatenation of the attributes $(score, docid)$ for candidate ordering.

Note that in particular keeping a large priority queue in memory at any time of the query processing may be expensive. Efficient implementations of the basic TA algorithm may diverge from the strict notion of constantly sorted queues. Alternative approaches may also use a bounded queue or merely keep all valid candidates in a single unsorted pool and iterate over that pool periodically, e.g., after large batches of sorted access steps or whenever needed to test the stopping condition.

Throughout the thesis, we will primarily use the IR-oriented terminology of documents and terms, but the proposed methods and results can be carried over to settings with numerical or categorical attributes of structured records or domain specific concept similarities in a very straightforward way.

1.4 Related Work

1.4.1 Top-k Query Processing

The state of the art on top-k queries over large disk-resident (inverted) index structures has been defined by the seminal work on *threshold algorithms* (TA) [Fag99, FLN01, FLN03, GBK00, GBK01, NR99]. TA scans all query-relevant index lists in an interleaved manner and aims to compute "global" scores for the encountered data items by means of a monotonic score aggregation function such as (weighted) sum, or maximum, etc. The algorithm maintains the worstscore among the current top-k results and the best possible score for all other candidates and items not yet encountered. The latter serves as a threshold for stopping the index scans when no candidate can exceed the score of the currently k^{th} ranked result. The algorithm comes in three variants: The original *TA-random* approach eagerly looks up all local scores of each encountered item and thus knows the full score immediately when it first encounters the item. Since random accesses may be expensive and, depending on the application setting, sometimes infeasible, the alternative *TA-sorted* method (coined NRA in [FLN01, FLN03] and Stream-Combine in [GBK01]) maintains *worstscore* and *bestscore* bounds for data items based on partially computed global scores, and its stopping test com-

pares the *worstscore* of the kth ranked result (typically coined *min-k*) with the *bestscore* of all other candidates. Hybrid approaches, such as the *Combined Algorithm* (CA) [Fag02], extend TA-sorted by a cost-model for a few carefully scheduled random accesses for the final scores of the most promising candidate items.

Obviously, TA-random is more effective in pruning the index scans and, thus, typically stops after a lower amount of overall index accesses than TA-sorted, but TA-sorted completely avoids expensive random accesses, and therefore, potentially can achieve better runtimes. TA-combined aims at minimizing the overall query cost with regard to a environment-specific cost ratio c_R/c_S of random versus sorted accesses and therefore is the most versatile approach for a wide range of system setups.

Numerous variants of TA have been studied for multimedia similarity search [dVMNK02, CGM04, NCS+01], ranking query results from structured databases [ACDG03], and distributed preference queries over heterogeneous Internet sources such as digital libraries, restaurant reviews, street finders, etc. [CwH02, MBG04, YSMQ01]. Marian et al. [MBG04] have particularly investigated how to deal with restrictive sources that do not allow sorted access to their index lists and with widely varying access costs. To this end, heuristic scheduling approaches have been developed, but the threshold condition for stopping the algorithm is a conservative TA-style test. Other top-k query algorithms in the literature include nearest-neighbor search methods based on an R-tree-like multidimensional index [ARSZ03, BBK01, CP00, HS99, HS03] and mapping techniques onto multidimensional range queries [BCG02] evaluated on traditional database indexes. In this context, probabilistic estimators for selecting "cutoff" values have been developed by [CP00, DR99, TFP03] and applied to multidimensional nearest-neighbor queries.

Most of the above TA-centric work has studied the original TA algorithm with eager random accesses. NRA and CA, on the other hand, with their overhead for the candidate bookkeeping have been regarded as the less attractive variant to which one would resort only under specific circumstances. However, with a large number of potentially very long index lists, NRA should actually be the method of choice; and with its cost-aware support for random access, CA offers the potential for the highest cost savings, but also the highest demand for an efficient implementation and cost estimation.

Threshold Algorithm

Algorithm 1 shows pseudo code for the original Threshold Algorithm (TA) [Fag99]. TA uses a basic round-robin heuristic for sorted access schedu-

Algorithm 1 Threshold Algorithm (TA).

```
1: TA(Index Lists L_i, Query t_i,...,t_m)
2: top-k := ∅;
3: candidates := ∅;
4: min-k := 0;
5: for all Index Lists L_i (i=1..m) do
6:   // Perform next sorted access to L_i in round-robin mode
7:   <d, s_i(d)> := L_i.getNext();
8:   score(d) := s_i(d);
9:   high_i := s_i(d);
10:  threshold := 0;
11:  for all ν=1..m do
12:    if ν ≠ i then
13:      // Perform random access for d's score s_ν(d) in L_ν
14:      score(d) += s_ν(d);
15:    end if
16:    threshold += high_ν;
17:  end for
18:  if score(d) > min-k then
19:    top-k.removeMinkItem();
20:    top-k.insert(d);
21:    min-k := top-k.getMink();
22:  else if threshold ≤ min-k then
23:    return top-k;
24:  end if
25: end for
```

ling over m input sources $L_1, \ldots, L_m$. The TA algorithm pursues a so called *document-at-a-time* model, i.e., each candidate object that is detected through a sorted access is eagerly looked up for its final score in all the remaining input lists for the query, and a final aggregated score is obtained directly. This way, no score uncertainty is induced by the algorithm, and no candidate queuing besides the intermediate top-k result objects is required. Then an initial threshold denotes the sum of the $high_i$ scores at the current scan position of each list L_i, which is an upper bound for all yet unseen candidates. If this threshold falls below the worstscore among all the top-k items, i.e., the score of the currently k^{th} ranked result, the algorithm may safely terminate, because no yet unseen candidate can exceed this threshold any more.

No-Random-Access Algorithm & Combined Algorithm

Algorithm 2 shows pseudo code for the No-Random-Access Algorithm (NRA) [Fag99] and the Combined Algorithm (CA) [FLN01]. NRA assumes only sorted access to the input sources and does not require all candidates to be fully evaluated at all query dimensions. In contrast to the TA algorithm, NRA pursues a so-called *term-at-a-time* model, i.e., it evaluates documents for each query condition separately and remembers partial results in

an intermediate data structure. Thus, it merely provides $worstscore(d)$ and $bestscore(d)$ guarantees for each candidate d based on where d has already been seen during the sequential scans, plus the local $high_i$ values at each of the candidate remainder dimensions in $\bar{E}(d)$, respectively. It therefore extends the TA algorithm by some notion of candidate bookkeeping (or queuing) and introduces the *min-k* threshold test as described beforehand.

To this end, [FLN01, GBK01] already proposed the NRA variant of TA, but occasional, carefully scheduled RAs can still be useful when they can contribute to major pruning of candidates. Therefore, [Fag02] also introduced the Combined Algorithm (CA) framework but did neither discuss any data- or scoring-specific scheduling strategies, nor experiments. CA, finally, is a hybrid algorithm and extends NRA by a rather inconspicuous but effective option of performing a limited amount of random accesses to look up the final scores for the currently best candidates according to a simple cost model, thus cautiously pruning the best candidates from the queue after each round of sorted accesses, using the invariant $c_R \cdot \#RA \leq c_S \cdot \#SA$, where $\#RA$ denotes the number of random lookups for data items d at individual index lists L_i, and $\#SA$ is the number of sorted accesses to inverted index structures. However, the sorted access scheduling between lists remains standard round-robin for the TA, NRA, and CA baseline algorithms.

Throughout the thesis, we will mostly adopt Fagin's notion of abstract query costs, i.e., the c_R/c_S ratio will drive our cost assumptions analogously to [Fag02]. Then the overall execution cost of a query is computed as $\#SA + c_R/c_S \cdot \#RA$. This cost model decides on the amount of iteratively scheduled random IOs.

In Chapter 6, we show that the basic CA scheduling approach can be improved three ways: 1) the ordering of candidates for which RAs are scheduled, 2) when they the RAs are scheduled, and 3) how to optimize the SA scheduling beyond the simple round-robing heuristic.

Index Access Optimality

The family of threshold algorithms have been proven to be *instance optimal* for any monotonous score aggregation function and under the assumption that the parameters m, i.e., the query dimensionality, and k, the number of retrieved items, are considered as constants. In particular, CA has been proven to be cost-optimal within a constant factor of $4m + k$ per query, regardless of the cost ratio c_R/c_S. In practice, however, the actual runtime optimality of TA and NRA strongly depends on the underlying assumption of index list access type supported. For realistic disk-based access models, CA clearly outperforms NRA, whereas TA is typically out of the question

Algorithm 2 No-Random-Access Algorithm (NRA) and Combined Algorithm (CA).

```
1: NRA/CA(Index Lists L_i, Query t_i,...,t_m)
2: top-k := ∅;
3: candidates := ∅;
4: min-k := 0;
5: for all Index Lists L_i (i=1..m) do
6:   // Perform next sorted access to L_i in round-robin mode
7:   <d, s_i(d)> := L_i.getNext();
8:   worstscore(d) += s_i(d);
9:   E(d) := E(d) ∪ {i};
10:  high_i := s_i(d);
11:  CA: Consider RA on L_ν for ν ∈ Ē(d) according to cost model
12:  for all ν ∈ Ē(d) do
13:    worstscore(d) += s_ν(d); //perform RA for d in L_ν
14:    E(d) := E(d) ∪ {ν};
15:  end for
16:  bestscore(d) := worstscore(d) + Σ_{ν∈Ē(d)} high_i;
17:  if worstscore(d) > min-k then
18:    d' := top-k.removeMinkItem();
19:    top-k.insert(d);
20:    candidates.remove(d);
21:    min-k := top-k.getMink();
22:    if bestscore(d') > min-k then
23:      candidates.insert(d');
24:    end if
25:  else if bestscore(d) > min-k then
26:    candidates.update(d);
27:  else
28:    candidates.remove(d);
29:  end if
30:  if topk.size() = k & bestscore(candidates.top()) ≤ min-k then
31:    return top-k;
32:  end if
33: end for
```

because of its random accesses overhead. Note that $4m + k$ can in fact mean a huge gap for typical IR queries already, with $m = 5$ keywords and $k = 20$ retrieved results.

Other Variants: MPro, Upper, and Pick

[CwH02] and [BGM02, MBG04] developed the strategies MPro, Upper, and Pick for scheduling RAs on "expensive predicates". They considered restricted attribute sources, such as non-indexed attributes or Internet sites that do not support sorted access at all (e.g., a streetfinder site that computes driving distances and times), and showed how to integrate these sources into a threshold algorithm. [MBG04] also considered sources with widely different RA costs or widely different SA costs, e.g., because of different network bandwidth or server load. Our computational model differs from these settings in that we assume that all attributes stored in the inverted lists (but not necessarily

auxiliary data structures such as phrase offsets) are indexed with support for both SA and RA and that all index lists are on the same server and thus have identical access costs. For our setting, MPro [CwH02] is essentially the same as the Upper method developed in [BGM02, MBG04].

Upper alternates between RA and SA steps. For RA scheduling, Upper selects the data item with the highest upper bound for its final score and performs a single RA on the attribute (source) with the highest expected score, with additional considerations to source-specific RA costs and eliminating "redundant" sources. This is repeated until no data item remains that has a higher upper bound than any yet unseen document could have; then SA are scheduled in a round-robin way until such a data item appears again.

[BGM02] also developed the Pick method that runs in two phases. In the first phase, it makes only SA until all potential result documents have been read, i.e., as soon as the bestscore that a yet unseen document could have is not larger as the current k^{th} largest partial score of an already seen document (which corresponds to *min-k* in our terminology). In the second phase, it makes RA for the missing dimensions of candidates that are chosen similarly to Upper taking the (source specific) costs of RAs and the expected score gain into account.

1.4.2 Approximative Top-k and Efficient IR

The idea of approximate top-k queries has been around both in the IR and DB literature (see, e.g., [BL85, PF95, DR99, CP00, AdKM01, ARSZ03]) for a noticeable time. The initial work on approximate inverted vector searches as proposed in [BL85] already implements a threshold algorithm for combining multiple inverted lists with an early stopping condition that is in fact very close to the NRA algorithm. This stopping condition has a heuristic nature, since it does not have a notion of candidate bookkeeping for partially evaluated candidates that may still qualify for the exact top-k results and is thus deemed to return approximative results only. [PF95] were among the first to consider vague queries over multiple streamed input sources, taking assumptions on the score distributions in the individual input streams and even correlations among scores into account in order to estimate input cardinalities and query costs.

However, in terms of analyzing how much is lost in result quality by the relaxation, the prior work either introduced control parameters that are difficult to tune or they are based on homogeneity assumptions for multidimensional data distributions. The main focus of the earlier work was image similarity search over color, texture, and contour feature spaces, where the assumptions may indeed be justified. Furthermore, the relaxation control

parameters (e.g., a distance slack factor) of these models were difficult to translate into user-perceived guarantees. In contrast, this thesis presents a principled approach to approximate top-k queries with *probabilistic guarantees* about the error relative to the "exactly top-k" queries, translatable into guarantees about query result precision and recall. Our approach can cope with heterogeneous distributions where the score variability may radically differ among different text terms or attributes of a semistructured dataset. In this scenario, we concentrate on algorithms that process index lists by sorted access only, as we are aiming at high-dimensional data spaces such as Web or XML documents where queries need to access a potentially large number of very long index lists and random accesses would be very expensive. Our approach allows much more aggressive index list pruning, compared to the original NRA method with sorted access only.

A small number of papers have considered how to minimize random access costs in TA when data sources vary in speed and selectivity [YSMQ01, CwH02, MBG04]. To this end, simple, histogram-based probabilistic estimators have been developed for making scheduling decisions (i.e., deciding on which source (i.e., dimension) the next random access should be made). None of this prior work has attempted a principled approach to probabilistic score prediction and result guarantees.

Efficient processing of index lists for ranked retrieval is an old topic in IR research [BL85, MZ96]. In this context, sorted-access-only is the rule of the game. The pruning techniques considered here (see also [AdKM01, SCC+01]) are heuristic in nature in that they trade off some loss in result quality (effectiveness in IR jargon) for speed without being able to predict and control the resulting effects (other than by experimentation). For the special but important case where the global score is a weighted sum of TF·IDF-based text relevance and link-based, and thus query-independent, authority, additional pruning heuristics have been developed in [BP98, LS03].

Quit & Continue

One of the most famous, approximate IR-style algorithms for combining inverted lists is the strategy known as *Quit* & *Continue* [MZ96]. The algorithm is amazingly similar to the later NRA algorithm and also sequentially scans a set of inverted lists in an interleaved manner, thus keeping a set of the currently best score accumulators – which are nothing else but partly evaluated candidate items in our notation – in-memory. To limit the amount of memory consumed by the accumulators, the algorithm comes in two separated phases: 1) the *Quit* strategy would simply quit query processing after a designated amount of $K \gg k$ accumulators has reached a non-zero score (at the possible

expense of poor retrieval results); and 2) the *Continue* phase may then decide to continue query processing to gain more information about the currently active accumulators but allows no new accumulators to be added into the memory. Both phases yield approximative results only, the result quality of the continue strategy depends on the exact scan depth until the algorithm terminates. In the extreme case, the Continue strategy could degenerate to a full merge algorithm and might still return approximative results only.

1.4.3 Rank-Join Optimization

When scanning multiple index lists (over attributes from one or more relations or document collections), top-k query processing faces an optimization problem: combining each pair of indexes is essentially an equi-join (via equality of the tuple or document ids in matching index entries), and we thus need to solve a join ordering problem [CK97, IAE04, LCIS05]. As top-k queries are eventually interested only in the highest-score results, the problem is not just standard join ordering but has additional complexity. [IAE04] have called this issue the problem of finding optimal rank-join execution plans. Their approach is based on a DBMS-oriented compile-time view: they consider only binary rank joins and a join tree to combine the index lists for all attributes or keywords of the query, and they generate the execution plan before query execution starts. An alternative, run-time-oriented approach follows the Eddies-style notion of adaptive join orders on a per tuple basis [AH00] rather than fixing join orders at compile-time. Then the query optimization for top-k queries with threshold-driven evaluation becomes a *scheduling problem*. This is the approach that we also pursue here.

The original scheduling strategy for TA-style algorithms is round-robin over all lists (mostly to ensure certain theoretical properties). Early variants also made intensive use of *random access (RA)* to index entries to resolve missing score values of result candidates, but for very large index lists with millions of entries that span multiple disk tracks, the resulting random access is about 20–20,000 times slower than sequential accesses that would be ensured if TA processed index lists only by *sorted access (SA)* in descending score order per list. This cost discrepancy results from the differences in sequential versus random disk I/O, additional CPU time for random access to appropriate data structures in memory, and the CPU overhead of crossing of file-system or DBMS interfaces.

The strategies developed in [CwH02, MBG04] for scheduling RAs on expensive predicates mostly concentrate on the issue of when to perform RAs but leave the schedule for SAs relatively straightforward. None of this prior work used any advanced score statistics; they merely built on information

about the scores they have seen during the scans (especially the score of the current scan position and the score gradient up to this point [GBK01]) and the average score in an index list.

RankSQL

The rank-aggregation framework [IAE03, ISA+04] and its latest outcome, RankSQL [LCIS05, LSCI05], provide a complete framework for the integration of rank-aggregation operators into relational database management systems (RDBMS), including query optimization, as well as adaptive plan costing. The rank-aggregation framework estimates the score-aware selectivity for binary top-k join queries, in order to compute and propagate a number of k' values for the subordinate top-k operators along a hierarchical tree of rank-aggregation operators for m-dimensional joins. In doing so, it merely assumes uniformly distributed scores for the basic input lists, but it provides an elegant approach that utilizes the Central Limit Theorem to estimate the top-level join-selectivity over a hierarchically arranged tree of binary top-k operators for processing conjunctive, m-dimensional queries.

Scrambling & Eddies

There have been two principal proposals in the literature that break the traditional design of query optimization and execution, namely Scrambling [AFTU96, UFA98] and Eddies [AH00, RDH03, DH04]. In Scrambling, scheduling the execution of query operators activates different parts of the query plan to adapt to the high latency incurred by remote data sources, for example, a wide area network. In general, the scrambling framework consists of two phases: a scheduling phase and an operator synthesis phase. The scheduling phase does not change the query plan structure; rather, it allows for different operators to be executed independently in their own execution threads. If an operator cannot proceed or experiences long execution delays, other operators in the plan are scheduled to execute. In the second phase, new query operators are formed when the original plan structure stalls or cannot produce any useful work.

The Eddies architecture and its variants continuously optimize a running query by routing individual tuples to the different query processing operators, thus eliminating the traditional query planning altogether. In contrast to traditional database engines, all the ranking information for a candidate object is encapsulated into an Eddy; no local rankings are kept among inputs. All the recently proposed extensions [RDH03, DH04] can be applied to minimize the runtime overheads of tuple routing. Principally, our cost-based

scheduling approaches can be cast into the notion of an Eddy, thus optimizing query executions for each potential top-k result candidate individually, taking both sorted and random access costs into account for scheduling.

In contrast to [AH00, IAE04], we do not restrict ourselves to trees of binary, conjunctive joins but consider index lists to be equally relevant to the query and are able to process m-dimensional, non-conjunctive queries in a single top-k-join operator.

1.4.4 Query Expansion

There is a rich body of literature on query expansion (see, e.g., [BSWZ03, BSA94, XC96, CTZC04, HCO03, Kwo04, LLYM04, MSB98, QF93, BSWZ03, Voo94, XC96]). All methods aim to generate additional query terms that are "semantically" or statistically related to the original query terms, often producing queries with more than 50 or 100 terms and appropriately chosen weights such as Rocchio [Roc71] or probabilistic Roberston & Sparck-Jones relevance weights [RJ76] (see also Section 3.2.1). Given the additional uncertainty induced by the expansion terms, such queries are usually considered as disjunctive queries and incur very high execution costs for a DBMS-style query processing [BZ04b, BZ04a]. The various methods differ in the sources that they exploit for inferring correlated terms: explicit relationships in thesauri, explicit relevance feedback, pseudo relevance feedback, query associations derived from query logs and click streams, summary snippets of Web search engine results, extended topic descriptions (available in benchmarks), or combinations of various techniques. In all cases, some similarity, correlation, or entropy measure between the original query terms and the possible expansion terms should be quantified (usually in some statistical manner), and a carefully tuned threshold needs to be determined to eliminate expansion candidates that are only weakly related to the original query. While such manual tuning is standard in benchmarks like TREC [TRE], it is almost black art to find robust parameter settings for real applications with highly dynamic corpora and continuously evolving query profiles [BZ04a] (e.g., in intranets, Web forums, etc.). This calls for *automatic* and *self-adaptive* query tuning.

Among the most successful expansion methods (at least in the TREC benchmark series) are probably the ones presented in [Kwo04] and [LLYM04]. [Kwo04] generate Google queries from the original query and use the summary snippets on the top-10 result page to generate alternative query formulations and performed very successful in recent TREC benchmarks. The final query expansion is a weighted combination of the original and the alternative queries. [LLYM04] uses a suite of techniques for extracting phrases and

word sense disambiguation (WSD), with WordNet [Fel98] as a background thesaurus and source of expansion candidates. Both of these techniques seem to require substantial hand-tuning for achieving their outstandingly high performance in result precision and recall.

1.4.5 XML IR

Efficient evaluation and ranking of XML path conditions is a very fruitful research area. Solutions include structural joins [AKJP+02], the multi-predicate merge join [ZND+01], the Staircase join based on index structures with pre- and postorder encodings of elements within document trees [Gru02] and Holistic Twig Joins [BKS02, JWLY03]; the latter, aka. path stack algorithm, is probably the most efficient method [CMW03] for twig queries using sequential scans of index lists and linked stacks in memory. However, it does not deal with uncertain structure and does not support top-k-style threshold-based early termination.

Vagena et al. [VMT04] apply structural summaries to efficiently evaluate twig queries on graph-structured data, and Polyzotis et al. [PGI04] present an efficient algorithm for computing (structurally) approximate answers for twig queries. [LYJ04] extends XQuery to support partial knowledge of the schema. None of these papers considers result ranking and query optimization for retrieving the top-k results, only.

Information retrieval on XML data has become popular in recent years. Some approaches extend traditional keyword-style querying to XML data [CMKS03, GSBS03, HPB03]. [FG01, CK01, TW00] introduced full-fledged XML query languages with rich IR models for ranked retrieval. [CMM+03] and [GS03] developed extensions of the vector space model for keyword search on XML documents. [SM02] addressed vague structural conditions, and [AYLP04] combined this theme with full-text conditions. More recently, various groups have started adding IR-style keyword conditions to existing XML query languages. TeXQuery [AYBS04] is the foundation for the W3C's official Full-Text extension to XPath 2.0 and XQuery [W3Ca]. [Feg04] extends XQuery with relevance ranking for keyword conditions and presents a pipelined architecture for evaluating queries but does not consider finding only the best results. [AKYJ03] introduced a query algebra for XML queries that integrates IR-style query processing.

TIX [AKYJ03] and TAX [JLST01] are query algebras for XML that integrate IR-style query processing into a pipelined query evaluation engine. TAX comes with an efficient algorithm for computing structural joins. The results of a query are scored subtrees of the data; TAX provides a threshold operator that drops candidate results with low scores from the result set.

TOSS [HDS04] is an extension of TAX that integrates ontological similarities into the TAX algebra.

Recent work on making XML ranked retrieval more efficient has been carried out by [KKNR04] and [MAYKS05]. [KKNR04] uses path index operations as basic steps; these are invoked within a TA-style top-k algorithm with eager random access to inverted index structures. The scoring model can incorporate distance-based scores, but the experiments in the paper are limited to DB-style queries rather than XML IR in the style of the INEX benchmark [INE], using a large annotated collection of IEEE Computer Society publications. [MAYKS05] focuses on the efficient evaluation of approximate structural matches along the lines of [AYBS04]. It provides different query plans and can switch the current query plan at runtime (i.e., the join order of individual tuples following ideas of [AH00]) to speed up the computation of the top-k results. The paper considers primarily structural similarity by means of outer joins, and disregards optimizations for content term search.

TopX also uses sorted index lists, but keeps a candidate queue in-memory and therefore is able to focus on sequential disk access and on minimizing random disk access through sophisticated index structures and judicious scheduling decisions. The prior work that is closest to the TopX engine is [KKNR04]; our performance studies in Section 9.8 compare TopX against this work.

XRANK

Among the most prominent IR approaches for ranked retrieval of XML data is XRANK [GSBS03]. It generalizes traditional link analysis algorithms such as PageRank [BP98] for authority ranking of linked HTML collections and conceptually treats each XML element as an interlinked node in a large element graph. Then the *element rank* of an XML element corresponds to the PageRank value computed over a mixture of containment edges, obtained from the XML tree structure, and hyperlink edges, obtained from the inter-document link structure similar to the HTML case. XRANK may indeed return deeply nested elements but merely supports conjunctive keyword search; it does not support structured query languages such as XPath [W3Cc]. For efficient retrieval of multi-keyword queries, it also uses inverted lists sorted in descending order of element ranks and sketches the usage of standard threshold algorithms such as TA [Fag99] for pruning the search space.

FleXPath

FlexPath [AYLP04] integrates structure and keyword queries and regards the query structure as templates for the context of a full-text keyword search. The query structure (as well as the content conditions) can be dynamically relaxed for ranked result output according to predefined tree editing operations when matched against the structure of the XML input documents. The FlexPath query processor already comprises the usage of top-k-style query evaluations for a slightly modified, XPath-like query language that later evolved as part of the W3C Full-Text extensions to XPath 2.0 and XQuery 1.0 [W3Ca]. Like [KKNR04], it uses separate index structures for storing and retrieving the structural and content conditions of a path query; it thus requires eager random access to disk-resident index structures for resolving the final structure of a result candidate.

XIRQL

XIRQL [FG01], a pioneer in the area of ranked XML retrieval, presents a path algebra based on XQL, an early ancestor of W3C's XQuery [W3Cd], for processing and optimizing structured queries. It combines Boolean query operators with probabilistically derived weights for ranked result output, thus transferring the probabilistic IR paradigm to the XML case. It defines data-type-specific vague predicates for vague search over differently typed XML elements such as person names or numbers, and introduces a notion of *index objects* that serve as an anchor from which the probabilistic weights are derived from (in the classic IR notion of a document). Using index objects follows the idea that only nodes of specific type and granularity in the document hierarchy should be presented as results to the end-user. Defining these objects, however, may be strongly schema-dependent and assumes substantial knowledge about the general document structure, e.g., derived from a preferably compact document type definition (DTD).

XXL

Finally, our group's prior work on XXL [TW00, TW02], specifies a full-fledged, SQL-oriented query language for ranked XML IR with a high semantic expressiveness that makes it stand way apart from the predominant XQL and XPath language standards. For ranked result output, XXL leverages both a standard IR vector space model and an ontology-oriented similarity search for the dynamic relaxation of structure and term conditions. The principal structure of the query, however, is evaluated in a strictly Boolean manner. For query processing, XXL does not use a top-k algorithm; TopX,

on the other hand, focuses on a small, XPath-like subset of the XXL query language which allows for a radically different query processing architecture that outperforms XXL in terms of efficiency by a large margin.

1.5 Contributions

1.5.1 Top-k Query Processing with Probabilistic Guarantees

The original TA method is conservative in that it stops scanning index lists only when it is certain that no more top-k results can be found. We believe that this is overly conservative given that the concept of a top-k query – especially in IR – has a heuristic nature anyway. Hardly any end-user would be interested in looking at exactly the k best matches to a similarity query. Rather, the rationale of top-k ranking is that users typically find one or a few relevant and novel data items among the top 10 or 20 results. So there is an inherent and unavoidable risk of missing the truly best results (in the subjective judgment of the user) anyway. This in turn justifies relaxing the concept of a top-k query into an approximate notion such that the query processor can occasionally tolerate errors: false positives or false negatives with regard to the top-k.

In this thesis, we develop various score predictors and show how to apply the derived predictors for a probabilistically relaxed and robust family of top-k algorithms, coined *Prob-k*. We provide probabilistic guarantees for the approximate top-k results in terms of (expected) precision and recall. Our experiments confirm that the probabilistically derived bounds are amazingly close to the empirically observed behavior.

To this end, we explore a variety of techniques including aggregate score predictors using histograms, efficiently evaluable Poisson estimations, and convolutions based on moment-generating functions with generalized Chernoff-Hoeffding bounds for the resulting tail probabilities. As the overhead of these techniques is crucial, the details of our bookkeeping and candidate testing strategies are all but straightforward; we explore a wide range of strategies within the paradigm of threshold algorithms based on different setups of priority queues.

1.5.2 Index Access Scheduling

Approximation may be a viable choice for a broad range of IR applications; for a database applications, however, this may not always be acceptable.

We show that similar probabilistic models as used for the *Prob-k* family of algorithms can in fact be used to improve index access scheduling decisions and, thus, greatly accelerate top-k queries with no loss in result precision.

The NRA and CA variants of Fagin's family of threshold algorithms perform sequential disk accesses for sorted index scans; CA also has the option of performing random accesses to resolve score uncertainty for a limited amount of promising candidates. The baseline SA scheduling in this work, however, is a simple round-robin heuristic and the RA scheduling follows a very basic cost model. This calls for a more elaborated investigation of the scheduling problem for these two kinds of accesses.

Improvements over Fagin's baseline algorithms have been sparsely studied in the literature, and only for each of the two access types in isolation. We take an integrated view of the scheduling issues and develop novel strategies that have the potential to outperform prior proposals. Our main contribution are new, principled scheduling methods based on a Knapsack-related optimization for sequential accesses and a probabilistic cost model for random accesses. The methods can be further boosted by harnessing probabilistic estimators for scores, selectivities, and index list correlations. We provide an integrated strategy that combines SA and RA scheduling in TA-style top-k query processing.

Our best combined scheduling strategies methods achieve significant performance gains compared to the best previously known method, namely Fagin's Combined Algorithm (CA). We also show that our best techniques are very close to an empirically computed lower bound for the execution cost of this class of threshold algorithms. The proposed scheduling decisions can be seamlessly integrated with the probabilistic pruning methods as mentioned before.

1.5.3 Dynamic & Self-tuning Incremental Query Expansions

We present a novel approach for efficient and self-tuning query expansion that is natively embedded into a top-k query processor with optional probabilistic candidate pruning and index access scheduling support. Traditional query expansion methods select expansion terms whose thematic similarity to the original query terms is above some specified threshold, thus generating a disjunctive query with much higher dimensionality. This poses three major problems:

1) the need for hand-tuning the expansion threshold,

2) the potential topic dilution with overly aggressive expansion, and

3) the drastically increased execution cost of a high-dimensional query.

Our key techniques for making query expansion efficient, scalable, and self-tuning are to avoid aggregating scores for multiple expansion terms of the same original query term and to avoid scanning the index lists for all expansion terms. For example, when the term "disaster" in the query "transportation tunnel disaster" is expanded into "fire", "earthquake", "flood", etc., we do not count occurrences of several of these terms as additional evidence of relevance. Rather, we use a modified best-match score aggregation function that counts only the maximum score of a document for all expansion terms of the same original query term, optionally weighted by the similarity (or correlation) of the expansion term to the original term. Furthermore and most importantly for efficiency, we open scans on the index lists for expansion terms as late as possible, namely, only when the best possible candidate document from a list can achieve a score contribution that is higher than the score contributions from the original term's list at the current scan position or any list of expansion terms with ongoing scans at their current positions.

Our novel Incremental Merge algorithm conceptually merges the index lists of the expansion terms with the list of the original query term in an incremental, on-demand manner during the runtime of the query. For phrase matching (i.e., adjacent or nearby words such as composite nouns), a novel technique for nesting top-k computations is used. For further speed-up, probabilistic score estimation can be used, considering score distributions and term selectivities.

We also investigate on various approaches for Word Sense Disambiguation (WSD) that can be carried over to the thesaurus-based query expansion methods we propose.

1.5.4 Efficient XML Full-Text Search

The thesis presents a novel engine for efficient ranked retrieval of XML documents over semistructured but non-schematic data collections. The algorithm follows the paradigm of threshold algorithms for top-k query processing with a focus on inexpensive sequential accesses to disk-resident index lists with only a few judiciously scheduled random accesses. We provide a prototype search engine for ranked XML retrieval, supporting the most important W3C XPath 2.0 Full-Text extensions as well as the complete NEXI specification used in the INEX benchmark series [INE], thus supporting search along all XPath axes with IR-style search conditions and optional thesaurus-based query expansion.

A typical example query could be phrased in the NEXI syntax as follows:

```
//book[about(.//, "Information Retrieval XML")
      ]//[
        about(.//affiliation, "Stanford")
          and
        about(.//reference, "PageRank")
      ]
```

This twig query should find the best matches for books that contain the terms "Information Retrieval XML" and have descendants tagged as affiliation and a reference with content terms "Stanford" and "PageRank", respectively. It should also find books about "statistical language models for semistructured data", and if no author from Stanford qualifies, it may provide books from someone at Berkeley as an approximate, but still relevant result. In addition, we may consider relaxing tag names such that, for example, monographies or even survey articles are found, however, with a lower score.

A viable solution must reconcile local scoring for content search conditions, score aggregation, and path conditions for joining the matches. As a key factor for efficient performance, it must be careful about random accesses to index structures. It should exploit precomputations as much as possible and may utilize the technology trend of fast growing disk space capacity (whereas disk latency and transfer rates are improving only slowly). The latter makes redundant data structures attractive, if they are selectively accessed at query run-time. It should be self-throttling and self-tuning with regard to thesaurus-based query expansion and expansion thresholds.

Prior work has addressed individual aspects of the above desiderata to some extent, but we are not aware of any comprehensive solution. The salient properties of TopX and our novel contributions for ranked XML retrieval are the following:

1) It efficiently processes XML IR queries with support for all XPath axes, carefully designed index structures, efficient priority queue management, and judicious scheduling of expensive random accesses to test both content-related and structural query conditions. It carries over our previous work for estimating aggregated scores of candidates as well as selectivities of both content conditions and structural path conditions to the XML case that drive our scheduling decisions.

2) It optionally supports probabilistic candidate pruning but considerably extends these prior techniques to an XML setting, in order to gain additional speed at the expense of a small loss in precision of the top-k results.

3) It is a highly versatile building block for a wide range of query models. It can be configured to support either strict conjunctions of search conditions or an "andish" retrieval mode, where it is allowed that some content or path conditions are not satisfied and merely result in lower scores. The result list can be either individual XML elements or entire documents. The scoring model takes into consideration both statistics about tag-term combinations in the underlying XML corpus and the compactness of subtrees that satisfy search conditions. It can easily be customized to specific kinds of scoring models depending on the application needs.

4) Finally, for thesaurus-based similarity search, it integrates our Incremental Merge technique that significantly improves prior work in terms of efficiency and query result precision. It efficiently supports "virtual" index lists that are not materialized but computed on demand using nested top-k operators for phrase expansions within XML elements.

The scoring model and query evaluation strategies applied in TopX achieved very promising results in the INEX benchmark 2005 [TS05].

1.5.5 Selected Publications

Selected aspects of this thesis have been published in several international conferences and workshops, namely:

- WebDB '03: *Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data.* Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 6th International Workshop on the Web and Databases, San Diego, CA, 2003.

- VLDB '04: *Top-k Query Processing with Probabilistic Guarantees.* Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 30th International Conference on Very Large Data Bases, Toronto, Canada, 2004.

- ER '04: *Towards a Statistically Semantic Web.* Gerhard Weikum, Jens Graupmann, Ralf Schenkel, and Martin Theobald. 23rd International Conference on Conceptual Modeling, Shanghai, China, 2004.

- SIGIR '05: *Efficient and Self-Tuning Incremental Query Expansion for Top-k Query Processing.* Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 28th Annual International Conference on Research and Development in Information Retrieval, Salvador, Brasil, 2005.

- VLDB '05: *An Efficient and Versatile Query Engine for TopX Search.* Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 31st International Conference on Very Large Databases, Trondheim, Norway, 2005.
- PKDD '05: *Word Sense Disambiguation for Exploiting Hierarchical Thesauri in Text Classification.* Dimitrios Mavroeidis, George Tsatsaronis, Michalis Vazirgiannis, Martin Theobald, and Gerhard Weikum. 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, 2005.
- VLDB '06: *IO-Top-k: Index-optimized Top-k Query Processing.* Holger Bast, Debapriyo Majumdar, Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 32nd International Conference on Very Large Databases, Seoul, Korea, 2006.

1.6 Overview of the Thesis

The rest of the thesis closely follows the structure denoted by the main building blocks described in the introductory sections. Chapter 2 defines our data model and the query language for text and semistructured data. It also defines our specific full-text extensions for ranked XML retrieval and the relational schemata used to store the various data formats. Chapter 3 defines our computational and scoring model with novel extensions for Web IR and XML IR. We also discuss common indexing versus query time trade-offs of dynamic retrieval environments. Chapter 4 presents the core TopX query processor. The implementation uses the same multi-threaded algorithmic skeleton for index scans, top-k bookkeeping, and candidate pruning for both text and semistructured data. Chapter 5 develops the probabilistic foundations for aggregated score predictors and convolutions, as well as different strategies for efficient candidate queuing and probabilistic candidate pruning. Chapter 6 extends this probabilistic machinery by selectivity estimators and feature correlations and integrates these advanced statistics into different cost models for sorted and random access scheduling. Chapter 7 discusses our novel Incremental Merge technique for dynamic and self-tuning query expansions. Chapter 8 extends the given infrastructure by a structure-aware top-k query processor, with efficient support for vague path query evaluations, XML-specific random accesses scheduling, and dynamic query expansion for content and structure. Chapter 9 presents a detailed experimental evaluation of the algorithms proposed. Chapter 10 concludes the thesis.

Chapter 2

Data & Query Model

2.1 Data Model

Libraries were among the first institutions to adopt and pursue the usage of IR systems. The first generation information systems were developed by academic institutions and later often replaced or taken over by commercial vendors. Among the first research-oriented library management systems that were developed for academic libraries and which are still in use today were Okapi (at City University London), MELVYL (University of California Libraries), and Cheshire II (at UC Berkeley).

In the early days of IR, these systems were mostly aiming to automate the functionality of card catalogs. They were largely restricted to searching for authors and titles, or using a small set of so-called *index terms* that had to be manually assigned to documents by a human expert. We will refer to these representative keywords also as the *logical view* of the document that abstracts from its physical structure and provides a compact representation of the document's entities and their relationships. With the increased capacity and performance of evolving computing systems, the second generation of IR systems came up with greatly advanced search functionality over an extended and automatically extracted set of index terms or even taking the full set of terms into account. We will refer to this extended logical view as the *full-text* representation of documents. The third generation of IR systems, that is currently evolving, increasingly abandons the classic notion of documents as primary retrieval units. They focus on advanced graphical interfaces and user guidance to support the search process, on question answering or multimedia retrieval, and on the adoption of tagged or semantically annotated text passages with multiple weighted fields or even hierarchically structured text contents in the form of XML data [W3Cb].

XML documents may contain highly-structured data (e.g., numbers, dates, etc.), unstructured data (untagged free-flowing text), and semistructured data (text with embedded tags). When a document contains unstructured or semistructured data, it is important to apply basic IR techniques such as scoring and weighting for concise retrieval. Recent standardized IR extensions to structured query languages such as the W3C XPath 2.0 Full-Text extension and the NEXI [TS04a, TS04b] query language, that is basically an outcome of the INEX [INE] benchmark series, reflect the increasing interest in ranked retrieval of semistructured data.

2.1.1 Text

In order to provide a compact representation of the logical view of a document, multiple occurrences of terms across a document are aggregated into an initial term score that aims to model the *relevance* of a term for the given document, thus ignoring the original ordering of terms in the document which is typically referred to as *bag-of-words model* in IR. Similarly, we may want to consider the frequency of a term across the whole corpus in order to capture its *specificity*. In this retrieval model, text passages are either considered to be unstructured, i.e., of equal importance, or they may be tagged with multiple weighted fields that influence the credit a term is payed for the document. For example, it might make a difference if a term occurs in the title or in the body of a document in order to determine its final score.

Typically, the index terms are extracted from the physical document structure automatically by tokenizing the text passages according to a set of white space delimiters (e.g., blanks, commas, line breaks, etc.), aggregated into term-score pairs (often called *document features*), and stored in the inverted index lists. In order to decrease the dimensionality of documents and queries and to support a simple form of content abstraction for vague search, standard IR techniques such as *stemming* and *stopword removal* are often applied in the initial tokenization step [BYRN99, MS99a, GF05].

Bag-of-Words Model

For determining the relevance of a query term t_i for a particular document d_j and the specificity of t_i across the collection, we keep basic statistics like

- the *term frequency* tf_{ij} of term t_i in document d_j, i.e., the number of occurrences of t_i in d_j, and
- the *document frequency* df_i of term t_i in the whole corpus, i.e., the number of documents that contain t_i.

Rather than the actual document frequency df_i, term specificity is often referred to as *inverse document frequency* $idf_i = \log(N/df_i)$, because most commonly in IR, the importance of a query keyword t_i for a given document d_j is considered to be proportional to the term frequency tf_{ij} and inversely proportional to the (logarithmically dampened) document frequency df_i with respect to the scoring model applied (see Section 3.1.1).

Term Proximity & Phrases

For more sophisticated queries, namely phrase matching and proximity-based scoring functions, the bag-of-words model is not sufficient. For matching phrases, the precomputed local scores (typically using the default per-term scoring model, thus ignoring offsets) are combined with an additional scoring component for the phrase proximity that also takes individual term positions and their distances in the document into account, either as a binary filter or in a true combined scoring approach, the latter inherently making the scoring function non-monotonous with regard to the precomputed per-term scores (see Section 3.1.2).

For top-k-style query processing, the term offsets may principally either be encoded directly into the inverted lists in addition to the local scores, with an individual entry for each term occurrence per document, or they may be stored in an auxiliary data structure that is accessed solely through random accesses. Note that the first option would make the sequential scans (and the main memory consumption) about an order of magnitude more expensive for any type of query, no matter if phrases are used in the query or not; the second option, however, would require a clever scheduling strategy to minimize to amount of expensive random accesses to perform the phrase and proximity tests. In order not to pay a blanket increase of sequential query costs, we decided to adopt the notion of *expensive predicates* for these phrase tests in combination with the random access scheduling option as described in Section 4.2.

TopX also implements additional full-text operators such as negation ($-$), mandatory query terms ($+$), and thesaurus-based query expansion ($\sim$). Efficient support for phrase tests, negation, and mandatory query terms is part of the TopX core query processor as described in Chapter 4; in Chapter 7, we present a novel and efficient approach for similarity search with dynamic query expansion, including high-dimensional phrase expansions.

2.1.2 Structured Data

Preference queries over multi-attribute search tasks, e.g., booking a hotel with multiple options for prize, location, quality, etc., or product catalogs with multiple product features often impose the search over *categorical attributes*, where good query results call for some kind of similarity search over near matches.

As opposed to text data, the fields for these categorical attributes, multimedia features, or named entities, such as $Year$, $Genre$, $Date$, etc., do not naturally impose a simple ranking (e.g., varying term frequencies as in the text case) that helps to distinguish or rank these fields. Even with global, IDF-like document statistics, score entries for the attributes basically degenerate to constants.

These relations can be stored as multi-attribute database-style flat tables or in some canonical schema using the concatenation of each individual attribute-value pairs as key (e.g., $Genre = Action$) for the inverted lists, all with the same perfect static score (e.g., $t_{ij} = 1 \; \forall \; d_j$).

Similarity Scores for Categorical Attributes

Vague search for categorical attributes is strongly application dependent, with approaches for similarity scores ranging from absolute differences, e.g., for prices, date and time entities, over geographical distances for locations, or to some notion of correlation-driven corpus statistics. These similarities may be either precomputed and directly materialized in the inverted lists for a small number of similar conditions, or be merged incrementally and on-demand in the sense of a potentially large query expansion (see Chapter 7), the latter with the advantage of selecting *context-aware* expansion terms and similarity weights for each search context individually.

Note that through the option of defining mutually nested structured objects (as given by the XML syntax introduced in the next subsection), the structure may become hierarchical, thus forming trees of structured objects, or even arbitrary graphs when links between objects are also taken into account. This poses a large variety of challenges for efficient indexing and retrieval and calls for the efficient support for new path query languages.

2.1.3 Extensible Markup Language (XML) 1.0

With the Web and digital libraries, the demand for more structured retrieval emerged. With the specification of the Extensible Markup Language (XML) 1.0 standard by the World Wide Web Consortium (W3C), a new way of merg-

ing text contents with structured attributes, or even a hierarchical arrangement of document contents and interlinked document graphs was smoothed. Although the XML specification originally aimed at providing a versatile format for data exchange, XML meanwhile has emerged as *the* default standard for semantically annotated and hierarchically structured, mutually dependent information contents.

Unfortunately – at least from an IR point-of-view – the amount of semantically meaningful, richly annotated XML data remains vanishingly small compared to the huge amount of poorly structured text data found on the Web today, or highly structured data hidden in the so called Deep Web. In the following, we merely report the most important syntactical concepts of XML that yield the foundations for our retrieval model over semistructured data, for a full specification of the XML syntax, see [W3Cb].

Each XML document has both a *logical* and a *physical* structure. Physically, the document is composed of units called entities, i.e., tags (or markup) and character data (CDATA). An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup.

Physical Structure

XML documents consist of intermingled markup and CDATA (i.e., start-tags, text, and end-tags). Tags have a name and are delimited by opening (<) and closing angle brackets (>). Start-tags may include a white-space-separated list of attribute-value declarations.

- *Markup* takes the form of start-tags, end-tags, or empty-element tags.
- All text that is not markup constitutes the *character data* (CDATA) of the document.

The document structure may be validated against an optional schema specification, the so-called document type definition (DTD), that provides a grammar for a certain class of XML documents and defines the way XML elements and CDATA sections may be nested. In addition, data types for the CDATA entities may be specified using the *XML Schema* language. We call a document *well-formed*, if its logical and physical structures nest properly. In addition, a document may be well-formed, if it has an associated document type declaration (DTD) or XML schema declaration, and if the document complies with the constraints expressed in it.

Note that XML markup also includes comments, document type declarations, processing instructions, XML declarations, or text declarations; none of these markup extensions is relevant for the data model applied in this thesis. We merely consider DTD or XML schema validations as an elementary functionality of the given parsing infrastructure; our indexing and querying model aims at a schema-oblivious view of the data and efficient support for a possibly diverse structure.

Logical Structure

- A data object is an *XML document*, if it is well-formed according to the above definition.
- Each XML document contains one or more *XML elements*, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, and may have a set of attribute specifications. Each attribute specification has a name and a value. The name in an element's end-tag must match the element type in the start-tag and is prefixed by "/".
- *Element Type Declarations*
  - An element type has *element content* when elements of that type must contain only child elements (no CDATA), optionally separated by white space.
  - An element type has *mixed content* when elements of that type may contain character data, optionally interspersed with child elements.

    In this case, the types of the child elements may be constrained by a DTD, but not their order or their number of occurrences.
- *Attribute Type Declarations*
  - *Attribute declarations* are used to associate name-value pairs with elements. Attribute specifications must not appear outside of start-tags and empty-element tags.
  - *Attribute list declarations* specify the name, data type, and default value (if any) of each attribute associated with a given element type. An attribute name must not appear more than once per element start-tag.

XML does not only provide interspersed markup and text data, but may provide a hierarchically nested element structure and thus arrange elements in the form of a *tree*. That is why elements are often referred to as *element nodes* and CDATA sections as *text nodes*. Text nodes are always leaf nodes. The usage of specific `idref` and XLink/XPointer [XPO] attributes that explicitly provide inner-document element references and intra-document links, respectively, thus providing basic facilities to connect arbitrary element nodes, may break this strict notion of a tree structure and in fact turn the XML data into an *element graph*, including cycles.

```
<article id="conference/vldb05/theobald">
   <title>
     An efficient and versatile
     engine for TopX Search.
   </title>
   <abs>
     We present a novel engine,
     coined TopX, for ranked
     retrieval of XML documents.
   </abs>
   <sec st="Introduction">
     <par>
       Non-schematic XML data…
     </par>
   </sec>
   <sec st="Related Work">
     <par>
       Efficient evaluation of XML
       path conditions…
     </par>
   </sec>
</article>
```

Figure 2.1: An XML example document.

Note that an XML document must start with a leading tag or root element, so each XML document confirms to its top-level element and vice versa. Hence, each XML document is also an XML element (ignoring document type declarations and processing instructions).

Document Order

There is an ordering defined on all the nodes in the document, called *document order*, corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the

document. Document order corresponds to a left-to-right, depth-first traversal of the document's node structure. Thus, the root node will be the first node in document order. *Reverse document order* is the reverse of document order. Note that document order corresponds to a simple preorder traversal of the document tree (see Section 2.3.2).

Simplified XML Data Model

Throughout the thesis, we will consider a simplified XML data model where idref/XLink/XPointer links are disregarded. Thus, every document forms a tree of nodes, each with a *tag* and a *content*. We treat attributes as children of the corresponding node. The content of a node is either a text string or it is empty; typically (but not necessarily) non-leaf nodes have element content and empty text content. The text content of mixed content elements is defined as the concatenation of the elements' CDATA sections including white spaces as-they-are. Name space declarations (i.e., domain-specific prefixes of element names) are treated as extended tag names. In the next subsection, we introduce a new notion of element full-contents that constitutes a central issue for our logical view of element contents and the IR techniques applied for the XML case for the rest of the thesis.

2.1.4 Full-Content Text Model

With each element node we can additionally associate its *full-content*, which is defined as follows:

Definition 2.1.1 (Element Full-Content) *The full-content of an XML element is the concatenation of all the element's descendants' text nodes in document order, including element whitespace separators for mixed content elements.*

Note that the concatenation of text nodes according to a document-order enumeration preserves the element ordering in the resulting full-content strings and allows also for phrase matching across element boundaries. Strict whitespace treatment across element boundaries may be crucial for some applications and collections, in particular with mixed content elements. For example, a common (merely squiggling) pattern in the INEX data like

```
<st>I<scp>ntroduction</scp></st>
```

would otherwise lead to strange effects for retrieval. IR-style scoring models may in turn break this ordering of terms and turn the full-content text nodes again into a bag-of-words.

Hence, in analogy to the text case, we make use of basic IR statistics that view the content or full-content of a node or entire document as a bag-of-words, and define measures like the following:

- The *term frequency*, $tf(t_i, n)$, of term t_i in node n, i.e., the absolute number of occurrences of t_i in the text content of n,
- the *full term frequency*, $ftf(t_i, n)$, of term t_i in node n, i.e., the absolute number of occurrences of t_i in the full-content of n, and
- the *element frequency*, $ef_A(t_i)$, of term t_i with regard to tag name A, i.e., the number of nodes with tag name A that contain t in their full-contents across the whole corpus.

Optionally, we may apply *stemming* and *stopword removal* to the XML text contents which may in turn affect the tf, ftf, and ef values. This way, we conceptually treat each XML element as an eligible retrieval unit (i.e., in the classic IR notion of a document), with its expanded full-text nodes as content.

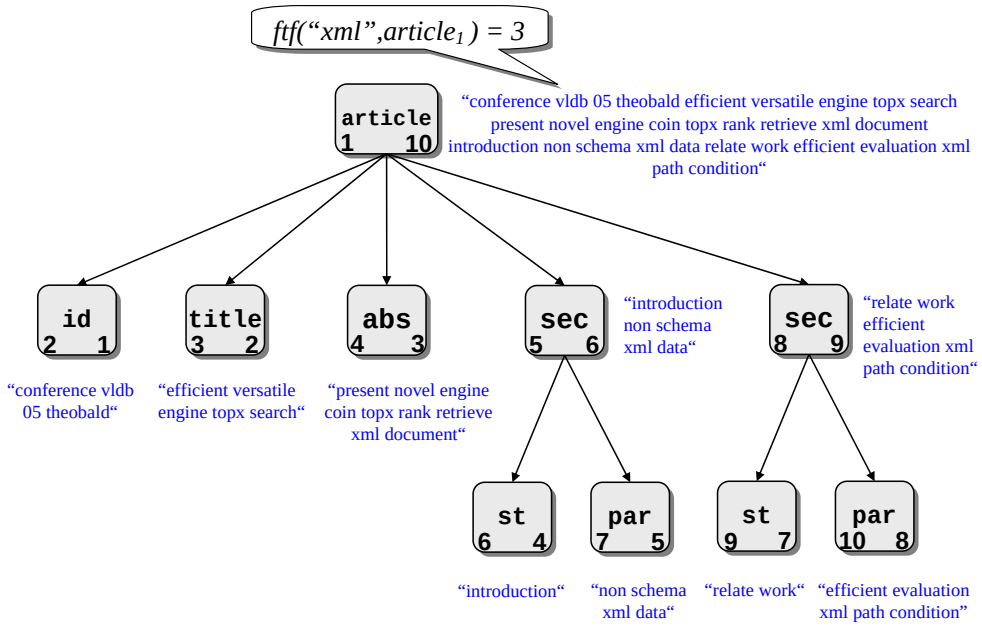


Figure 2.2: Redundant full-content text nodes for elements.

Figure 2.2 shows the ftf value of the term "xml" for the top-level article element as $ftf("xml", article_1) = 3$, whereas the plain text $tf("xml", article_1)$ value would be 0, because there is no text node connected to the

article as a direct child node. Nethertheless, the ftf value of 3 indicates some form of relevance that this particular article should be credited when searched for the term "xml". However, the whole `article` element might indeed be *more exhaustive* but *less compact* than one of the respective descendant elements like `sec` or `par` which should be reflected by the final scoring model that we apply for ranked retrieval.

2.2 Query Language

2.2.1 XPath 1.0

As for specifying the syntax of structured queries, we focus on the W3C core XPath 1.0 specification (see [W3Cc] for a full specification). The primary purpose of XPath is to address and select parts of an XML document. In support of this primary purpose, it also provides basic facilities for the manipulation of strings, numbers and Booleans in the form of predicates and functions. XPath uses a compact, non-XML syntax to facilitate the use of XPath within Unified Resource Identifiers (URIs) and XML attribute values. It operates on the abstract logical structure of an XML document, rather than on the surface syntax.

In addition to its use for addressing, XPath is also designed such that it has a natural subset that can be used for matching (i.e., testing whether or not a node matches a pattern). XPath models an XML document as a *tree of nodes* with various axes to support different types of nodes, including element nodes, attribute nodes, and text nodes.

Location Steps

XPath expressions are constituted of so-called *location steps*. The syntax for a location step is the *axis name* and *node test* separated by a double colon, followed by zero or more *predicate* expressions each in square brackets. For example, in

```
descendant::par[position()=3] ,
```

`descendant::` is the name of the axis, `par` is the node test, and `[position()=3]` is a predicate. The node set selected by the location step is the node set that results from generating an initial node set from the axis and node test, and then filtering that node set by each of the predicates in turn.

The initial node set consists of the nodes having the relationship to the context node specified by the axis, and having the node type and name

specified by the node test. For example, a location step `descendant::par` selects the `par` element descendants of the context node. `descendant::` specifies that each node in the initial node set must be a descendant of the context node; `par` specifies that each node in the initial node set must be an element named `par`.

Location Paths

There are two kinds of location paths that form an XPath expression: relative location paths and absolute location paths. A relative location path consists of a sequence of one or more location steps separated by "/". The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. The sets of nodes identified by each step are unioned together.

An absolute location path consists of "/" optionally followed by a relative location path. A "/" by itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.

Every location path can be expressed using a straightforward but rather verbose syntax. Besides the full syntax specification, there are a number of syntactic abbreviations that allow common query formulations to be expressed more concisely. This subsection will explain the semantics of the most important XPath axes using the unabbreviated syntax and giving abbreviation examples.

Navigational Axes

There are thirteen distinct XPath axes specified in XPath 1.0, with partly overlapping constraints on the structure. An axis is either a forward axis or a reverse axis. An axis that contains only the context node or nodes that are after the context node in document order is a forward axis. An axis that contains only the context node or nodes that are before the context node in document order is a reverse axis. Forward and reverse axes can be rewritten symmetrically by exchanging the source and target node tests.

Combining symmetric cases, we identify the following primary (non-redundant and indispensable) axes for path navigation, namely the child, self, descendant, descendant-or-self, following, following-sibling, and attribute axis (ignoring namespace declarations at this point). These seven axes do not overlap, and altogether they capture the full expressiveness of XPath:

- The *child axis* (`child::`) contains the children of the context node.

- The *self axis* (`self::`) contains just the context node itself.
- The *descendant axis* (`descendant::`) contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes.
- The *descendant-or-self axis* (`descendant-or-self::`) contains the context node and the descendants of the context node.
- The *following axis* (`following::`) contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes.
- The *following-sibling axis* (`following-sibling::`) contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty.
- The *attribute axis* (`attribute::`) contains the attributes of the context node; the axis will be empty unless the context node is an element.

The location path `attribute::idref` selects all attributes with the name `idref` for the context node; it is abbreviated as `@idref`. The most important abbreviation is that `child::` can be omitted from a location step. In effect, child is the default axis. For example, a location path

`sec/par`

is short for `child::sec/child::par`; and the widely known, mnemonic abbreviation for the descendant-or-self axis is `//`.

Node Tests

The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The meaning of some node tests is dependent on the axis. The axis affects how the expression in each predicate is evaluated and so the semantics of a predicate is defined with respect to an axis.

A qualified name in the node test is expanded into an expanded-name using the namespace declarations from the expression context. A node test that refers to a qualified name is true if and only if the type of the node is the principal node type and has an expanded-name equal to the expanded-name specified by the qualified name. For example, `/par` selects the `par` element

children of the context node; if the context node has no `par` children, it will select an empty set of nodes. `@idref` selects the `idref` attribute of the context node; if the context node has no `idref` attribute, it will select an empty set of nodes.

A node test using the wildcard expression `*` is true for any node of the principal node type (referring to an element or attribute node). For example,

```
//*
```

will select all element descendants of the context node, and `@*` will select all attributes of the context node. The node test `text()` is true for any text node. For example, `/text()` will select the text node children of the context node.

Predicates & Functions

Note that expressions of any complexity can be specified in square brackets which must be satisfied before the preceding node will be matched by an XPath. Among the most common predicates that are used in square brackets to restrain the node test in an expression is the `position` function. The proximity position of a member of a node set with respect to an axis is defined to be the position of the node in the node-set ordered in document order, if the axis is a forward axis, and ordered in reverse document order, if the axis is a reverse axis. The first position is 1. The most commonly used abbreviation in XPath is used for position tests of the form `//par[position()=3]` as

```
//par[3]
```

which selects the third `par` descendant of the context node in document order. A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the predicate expression is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position. Note that XPath already defines a large variety of string manipulating functions (e.g., the `contains` function); these are non-IR functions and merely provide a basic string matching facility without any ranking or similarity search designated.

Support & Target Elements

For simplicity, we only consider XPath expressions that form *query trees* at this point, although it is possible to formulate graphs and even cycles in XPath. Then, evaluating an XPath expression against an XML document

tree becomes a *tree matching* problem, thus finding a valid embedding of the XPath tree in a given document tree with the known polynomial complexity of $O(|D|^5\ |Q|^2)$ [GKP02, GKP03] (see also Section 8.3.5). Note that there does not necessarily have to be a unique embedding of the query tree into the document tree.

For IR-style evaluations of these path expressions, we distinguish two classes of node tests for retrieval, namely the query *target element* and the *support element(s)*, in order to define the result of an XPath query:

Definition 2.2.1 (Target Element) *The rightmost, top-level node test in a location path is called the target element of that path query.*

Definition 2.2.2 (Support Element) *All elements denoted by a path query that are not the query's target element are called the support elements.*

Definition 2.2.3 (Query Result) *Valid results of a path query are those XML elements that match the query's target element.*

For example, the XPath query

```
/article[/abs[contains(text(), "XML")] and
         //par[contains(text(), "path")]
        ]/title[1]
```

selects the first `title` element that is a child node of an `article` element that has a child node of type `abs` that contains the term "XML" and with `article` having an arbitrary descendant node of type `par` containing the term "path". Here, `title[1]` is the target element of the XPath query, and only this element should be returned by the XPath engine. Note that omitting the node predicate `[1]` may select more than one `title` element from an XML document if these are present in the document's structure. In doing so, the above XPath query would successfully return the (only) `title` node of the example document of Figure 2.1.

According to the above definition, a path query denotes exactly one node test as the target element of the query, although a document instance may yield more than one element that matches the path expression and, thus, more than one element that matches the query's target element. Note that the tag list syntax `//(A|B)` allowed in the XPath (and NEXI, see Subsection 2.2.3) language is an exception to this uniqueness constraint of the target element in the query, since it means "either the A or the B element" and, thus, provides a means to specify multiple target elements in a path expression, although this is hardly used in practice.

2.2.2 XPath 2.0 – Full-Text Extension

The W3C XPath 2.0 and XQuery 1.0 Full-Text extension [W3Ca] is probably the best-known effort to provide a standardized syntax for IR extensions in structured query languages. The full-text extensions are the W3C's reaction to the rising demand for a standardized way to support full-text search as well as structured search against XML documents. Note that a similar requirement for full-text search led ISO to define the SQL/MM-FT standard. SQL/MM-FT defines extensions to SQL to express full-text searches providing similar functionality as does the full-text language extension to XQuery 1.0 and XPath 2.0.

"FTContains" Operator

The central language construct for full-text search is the new `ftcontains` full-text operator whose semantics reaches way beyond the simple string manipulation functions provided in XPath 1.0. Beneath some "syntactical sugar" like phonetic similarities, the W3C full-text extension explicitly specifies the usage of case sensitivity, stemming, and stopword removal. Moreover, it is already designed to support user defined query weights and a notion of result scoring. For example, the XPath query

```
/article[
  ./title ftcontains ("xml" with stemming weight 0.8)
                  && ("java" weight 0.2)
]/author
```

returns the author of an `article` with a `title` containing a word with the same root as "xml" and the word "java", with "xml" weighted four times higher than "java". `ftcontains` also allows for the specification of a maximum distance for words in phrases and even thesaurus lookups for related terms. Recall that valid results for an XPath-like location path are only those element instances that match the query's target element; in a strict evaluation of the query's structure, these will be the only elements that may be assigned a positive score.

2.2.3 Narrowed Extended XPath I (NEXI)

TopX currently supports the full NEXI specification as defined in [TS04a, TS04b] which is intended to provide a compact and yet extended IR search functionality to XPath 1.0. In particular, the navigational axes have been truncated down to the descendant-axis alone, whereas a new `about` operator

for ranked element retrieval was added. Note that unlike in XPath, the most common axis abbreviation `//` in NEXI refers to the descendant axis, only.

"About" Operator

Similarly to the `ftcontains` operator in XPath 2.0 Full-Text, NEXI specifies an `about` full-text operator, but otherwise retains only a strongly reduced subset of the original XPath syntax. For example,

```
//article[about(.//title, +xml java)]//author
```

is probably the nearest match for the aforementioned XPath 2.0 Full-Text query in the NEXI syntax. While most of the explicit syntactic options that `ftcontains` provides are missing in NEXI, the semantics of the `about` operator by default denotes text contents to be interpreted as *vague*, i.e., they may be dynamically relaxed in an "andish" manner or at the same time be expanded through similar terms. The exact interpretation of the `about` function is left for each specific implementation. Moreover, NEXI specifies the usage of a + operator for mandatory terms, a − operator for excluded terms, and the usage of phrases denoted by quotation marks (`"`).

We extend NEXI to explicitly support an additional ∼ operator for the expansion of tags and content terms into similar expressions (according to semantic similarities, e.g., from a common-sense or user-defined thesaurus) for selected tags and terms, only. TopX implements a query rewriter that translates the NEXI syntax into the internal DAG-based representation of the query processor (see Subsection 8.2.1) and supports dynamic query expansion and similarity search using thesaurus lookups for these explicitly marked expansions.

In the following, we will often abbreviate the syntax of the `about` operator, e.g., using `title["xml java"]` as short for `//title[about(.//, "xml java")]`, thus referring to a vague interpretation of the text contents "xml" and "java". Similarly, we focus on the descendant axis (i.e., the full-content case) as the much more important case for XML IR with vague search, thus following the NEXI specification; the case for the child axis follows analogously.

Note that, if the target element of the query is interpreted as vague or even skipped completely as it is the case in a pure keyword or so-called content-only (CO) query, a good scoring model should tend to automatically apply a higher score for the more compact retrieval unit. Moreover, these local scores should be aggregated in such a way, that the most suitable retrieval unit for a *multi-dimensional query* is automatically returned at a higher rank than a less specific top-level element. Ideally, there should no benchmark- or

collection-specific tuning be necessary, thus making the manual preselection of commonly retrieved elements (such as sections or paragraphs in INEX) or the use of predefined retrieval units obsolete. These advanced considerations will become manifest in the scoring model described in Section 3.4.

Content-only Queries (CO)

Content-only (CO) queries correspond to traditional keyword queries in IR, containing only words and phrases; no structural constraints are allowed. CO queries can be rewritten into a valid XPath or NEXI expression using the wildcard node test `//*`. This way, any element in the corpus containing one or more of the keywords specified in the query is a potential match. This kind of query is often used when the user is unfamiliar with the structure of the XML collection, or does not know where exactly a relevant match could be found.

Content & Structure Queries (CAS)

Content and structure (CAS) queries contain explicit structural requirements. They arise if the user is aware of the structure and wants to specify either the granularity of the result element (e.g., whole articles versus compact paragraphs), or wants to explicitly constrain the way the query structure is matched against the XML collection.

In a *strict* interpretation (SCAS) of the target structure, the information is assumed to be exactly deducible from the query structure and only those elements are returned that exactly match *all* the support and target elements specified by the query. In a *vague* interpretation of the query structure, however, this strict notion of query results may be relaxed. Variations of this scheme include the interpretation of the support elements as vague and the target element as strict (SVCAS), or the interpretation of the support elements as strict and the target element as vague (VSCAS), and vice versa.

Specifying an information need by an exact structure is not an easy task, especially for a heterogeneous collection or a federation of XML documents from different sources. Often, it is more beneficial to dynamically relax the structural constraints if important content conditions cannot be matched otherwise, and to define a ranking for the structure, too. Then the score aggregation decides about the final result ranking of different element types. Therefore, TopX provides also structural scores for tag sequences or branching path queries and ranks results according to the combined score for both content and structure. Then the user can specify a preference for matching the structural or rather the content-related parts.

2.3 Relational Schemata for Text and Semistructured Data

Many path index structures for XML have been proposed in the literature (see, for example, [MS99b, LM01, CSF+01]). We believe that the XPath Accelerator [Gru02] and DataGuides [GW97] are among the simplest, most compact, and yet most effective approaches. Therefore, we adopt these two data structures for the TopX database schema, thus taking advantage of their (partly complementary) salient properties.

2.3.1 Text Schema

Transferring the bag-of-words model into a relational schema and, thus, storing text data in a relational database system is very straightforward. For each document, term, and score, we create triplets of the basic form $(docid, term, score)$, where the pair $(docid, term)$ is primary key. Efficient sorted and random access on top of the DBMS is supported by two B^+-trees on the attributes concatenated in $(term, docid, score)$ for SA and in the order (*docid*, *term*, *score*) for RA (see Appendix A.1.1 for table definitions). We refer to this base table as `TextFeaturesRA`. Section 8.2.2 discusses some tuning tricks for storing these index structures space-efficiently in Oracle.

Note that an additional offset index `TermsRA` is required for phrase matching with a B^+-tree index over attributes concatenated in the order (*docid*, *term*, *pos*), which is solely accessed through random lookups; see also Section 4.2 for an efficient scheduling approach of these phrase tests.

2.3.2 Structural Indexes for XML

XPath Accelerator

The default TopX method for testing path constraints and matching element regions is leveraging the pre-/postorder tree labeling scheme of Grust's XPath accelerator [Gru02, GvKT03], which is one of the simplest and yet most effective tree index structures for mapping XML data onto a relational schema. The XPath accelerator is specifically designed to support *all* 13 XPath axes and to run on top of a relational backend to leverage its stability and scalability.

In order to find a unique labeling scheme for the nodes of an XML document tree, we provide the following definitions:

Definition 2.3.1 (Preorder Traversal) *In a preorder traversal, a tree node*

v is visited and assigned its preorder rank $pre(v)$ before its children are recursively traversed from left to right.

Definition 2.3.2 (Postorder Traversal) *In a postorder traversal, a tree node v is visited and assigned its postorder rank $post(v)$ after all its children have been traversed from left to right.*

XPath-like location paths are split into single node tests for each tag condition individually, with pre/postporder comparisons for the path structure. Testing two nodes v and v' for their descendant relation then simply resolves in comparing their pre- and postorder ranges:

$$\begin{aligned} & v' \textit{ is a descendant of } v \\ & \Leftrightarrow \\ & pre(v) < pre(v') \quad \wedge \quad post(v') < post(v) \end{aligned} \tag{2.1}$$

These range tests for the pre- and postorder attributes naturally map to DBMS-style range scans for the $(pre, post)$ attribute pairs, coined *Staircase joins* in [GvKT03]; the initial paper also sketches an efficient R-tree [Gut84, KF93] support for the triplets of the form $(pre, post, level)$, thus implementing the full range of XPath axes.

Figure 2.3 depicts the pre/postorder coordinates of elements for the example document of Figure 2.1. The light-gray shaded area denotes the pre/postorder range for all descendants of node sec_5 which are nodes st_6 and par_7, respectively. The dark-gray area in turn depicts all descendants of node st_5 which is empty in this case.

TopX performs a variant of these Staircase joins for each candidate document in-memory, whenever additional elements have been detected through a sequential or random access for that candidate, and the path query is evaluated against the updated element structure of the candidate document (see Section 8.3 for details on the XML query processing). Using tag sequences instead of whole location paths has several advantages for our query evaluations. It provides the basis for flexible scorings for both content and structure and offers the possibility to dynamically relax structural query conditions similar to content conditions in the text retrieval case. Hence, it allows us to compensate some weak structural matches, if some node tests fail, for good matches in other query conditions, thus translating the advantage of non-conjunctive score aggregations to XML IR. And it inherently supports the NEXI-style usage of the descendant axis well.

Unfortunately, splitting a location path into its tag sequences also reveals some drawbacks for path expressions with very low selectivity, namely when

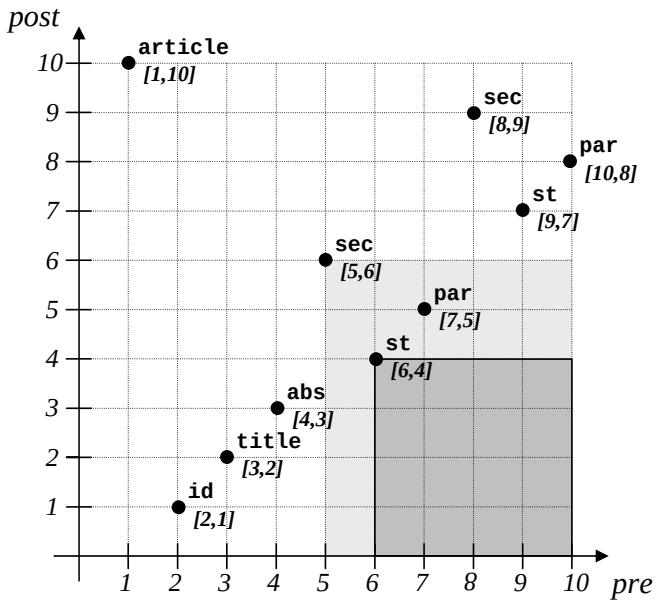


Figure 2.3: Pre/postorder ranges for all elements of the example document of Figure 2.1.

the structure is not matched for most candidates and many node tests fail which will incur an increased amount of node tests and typically increased random disk I/O. Optimizing path queries with a low selectivity is addressed by the next index structure, the DataGuide.

DataGuides

DataGuides [GW97] are another way to map structural information about XML data onto a relational schema. This index structure benefits from the observation that XML collections often exhibit regular patterns in the structure or follow a comprehensive schema definition, for example in the form of a compact DTD. DataGuides aim at providing concise summaries for the structure of a semistructured database. The key idea is that the schema can be represented as a *deterministic finite automaton* (DFA), the DataGuide, that accepts all distinct labeled path instances of the source database. The resulting DFA then provides an in-memory summary of the path structure of the source database. For a tree database, the conversion to the DFA takes linear time; for a graph, however, space and time consumption may become exponential.

The work initially presented in [GW97] uses path-to-bucketid mappings by enumerating all distinct root-to-leaf paths as captured by the DFA. Then

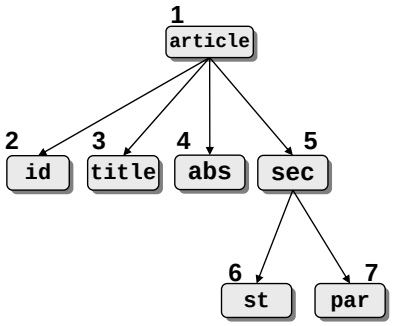


Figure 2.4: DataGuide for the example document of Figure 2.1.

Path	b-id
/article	1
/article/id	2
/article/title	3
/article/abs	4
/article/sec	5
/article/sec/st	6
/article/sec/par	7

Figure 2.5: DataGuide-like path-to-bucketid mappings for all paths of the example document of Figure 2.1.

each resulting bucket id represents a whole location path and can be matched against a path query. Table 2.5 shows a respective DataGuide structure for the XML example document of Figure 2.1, together with the resulting path-to-bucketid mappings. A simple extension is to assign distinct bucket ids for all the path prefixes, too.

DataGuides are generally the method of choice for indexing collections with a hierarchical, deeply nested element structure and a generally low selectivity of path queries. Unfortunately, using path summaries inherently supports only the child axis efficiently. Although it is principally feasible to support the descendant axis for DataGuides as well, this might render the original DataGuide automaton non-deterministic, with potentially high conversion and storage cost for the respective DFA.

Moreover, DataGuides do not provide information about the actual data instances, but rather serve as a structural filter. Joining individual elements on the basis of bucket ids only (e.g., for branching path queries) would cause the danger of returning false positives. Some kind of unique element identifier is required in this case, e.g., using the well-known preorder labels as described in the previous subsection that help us to unambiguously identify an XML element. We will revisit this issue in Section 8.5.2 when the efficient usage of hybrid index structures is discussed.

2.3.3 Combined Inverted Block-Index for XML

State-of-the-art systems for ranked retrieval based on these structural indexes typically conduct query processing on a combination of inverted lists for XML contents and structural indexes for the path evaluation *separately* (see, e.g., [KKNR04, AYLP04]), often with an eager policy for using random accesses to validate the structural part. An intriguing idea would be to trade off some inexpensive disk space to implement a *combined schema* for content and structure that helps to safe or at least drastically reduce the amount of random accesses necessary to resolve a candidate's structure. The latter approach, as pursued by TopX, offers great potential cost savings and increased retrieval efficiency, but also calls for novel solutions that can dynamically switch between the most suitable index structure and even employ hybrid indexes whenever cost beneficial. The basic TopX schema, which is one of our key approaches for efficient query evaluations on XML data, is described in this subsection.

Inverted index lists are stored as database tables; Figure 2.6 shows the corresponding schema definitions with some example data for three tag-term pairs. The current implementation uses Oracle 10g as a backbone, mainly for easy maintenance of the required index structures, whereas the actual query processing takes place on top of the DBMS exclusively in the TopX query engine, such that the DBMS itself remains easily exchangeable.

Nodes in XML documents are identified by the combination of document identifier (*docid*) and their preorder (*pre*) label. Navigation along all XPath axes is supported by both the *pre* and *post* attributes following the XPath accelerator technique of [Gru02]. Additionally, the *level* information may be stored to support the child axis as well, but it may be omitted for the NEXI-style exclusive usage of the descendant axis.

Content Index

The base table contains the actual node contents indexed as one row per tag-term pair per document, together with their local scores (referring either to the simple content or the full-content scores) and their pre- and postorder numbers. For each tag-term pair, we also provide the *maximum score* among all the rows grouped by tag, term, and document identifier to extend the previous notion of single-line sorted accesses to a notion of *sorted block-scans*. The actual index lists are processed by the top-k algorithm using two B^+-tree indexes that are created on this base table: one index for sorted access support in descending order of the $(maxscore, docid, score)$ attributes for each tag-term pair, and another index for random access support using

$(docid, tag, term)$ as key.

Each sequential block scan prefetches all tag-term pairs for the same document id in one shot and keeps that block of elements in memory for further processing which we refer to as *sorted block-scans*. Random accesses to content-related scores for a given document, tag, and term are performed through small range scans on the respective B^+ tree index using the triplet $(did, tag, term)$ as key. Note that grouping tag-term pairs by their document ids keeps the range of the pre/postorder-based in-memory structural joins small and efficient. All scores in the database tables are precomputed when the index tables are built.

sec[xml]

eid	docid	score	pre	post	max-score
46	2	0.9	2	15	0.9
9	2	0.5	10	8	0.9
171	5	0.85	1	20	0.85
84	3	0.1	1	12	0.1

st[introduction]

eid	docid	score	pre	post	max-score
216	17	0.9	2	15	0.9
72	3	0.8	10	8	0.8
51	2	0.5	4	12	0.5
671	31	0.4	12	23	0.4

par[schema]

eid	docid	score	pre	post	max-score
3	1	1.0	1	21	1.0
28	2	0.8	8	14	0.8
182	5	0.75	3	7	0.75
96	4	0.75	6	4	0.75

Figure 2.6: Inverted block-index with precomputed full-content text scores for tag-term pairs.

Figure 2.6 depicts some example data of the inverted block index for the three tag-term pairs `sec[xml]`, `st[introduction]`, and `par[schema]`. The gray shaded cells denote the element blocks for the document with id 2; the two gray marked rows for `sec[xml]` indicate that document d_2 has two `sec` elements that contain the term "xml": one with a local score of 0.9, and one with a local score of 0.5. Then the *maxscore* attribute of d_2 for the tag-term pair `sec[xml]` has a value of 0.9.

Note that a unique element identifier *eid* does not actually have to be materialized and may be generated on-demand as a linear combination of a bit-shift for the *docid* and the *pre* attribute, e.g., using $eid = (docid << 8) + pre$, whenever needed. Further note that keeping the document identifier *docid* in the inverted index helps us to limit the range of the in-memory structural joins for query processing which keeps the pre- and postorder comparisons efficient and enables us to dynamically switch between documents and elements as result granularity. Details for the query processing over this block structure will be given in Section 8.3.

For search conditions of the form `A[.//"t`$_1$ `t`$_2$`"]` referring to the descendant axis, we refer to the full-content text scores, based on $ftf(t_1, A)$ and $ftf(t_2, A)$ values of entire document subtrees; these are read off the precomputed base tables in a single efficient sequential disk fetch for each document until the *min-k* threshold condition is reached and the algorithm terminates.

We fully precompute and materialize this inverted block index to efficiently support the descendant axis between tag-terms pairs for typical NEXI query patterns of the type `A[.//"a"]`. With this specialized setup, parsing and indexing the INEX collection of about 17,000 large documents takes about 80 minutes on an average server machine including our XML-specific scoring model and the materialization of the inverted block-index view.

We propagate, for every term t that occurs in a node n with tag A, its local tf value "upwards" to all ancestors of n and compute the ftf values of these nodes for t. For search conditions of the form `A[.//"a1 a2"]`, referring to the descendant axis between the `A` tag and the term conditions `a1` and `a2`, we can efficiently precompute these full-contents scores, based on ftf values of entire subtrees which greatly helps accelerating query processing for these basic patterns.

The redundant full-content indexing introduces a factor of redundancy for the textual contents that approximately corresponds to the average nesting depth of text nodes of documents in the corpus (which corresponds to factor of 4-5 for the INEX collection); it is our intention to trade off a moderate increase in inexpensive disk space for faster query response times. Note that by using tag-term pairs for the inverted index lookups, we immediately benefit from more selective, combined tag-term features and shorter index lists for the textual contents, whereas the hypothetical combinatorial bound of $\#tags \cdot \#terms$ entries has by far not been reached for any of the collections we investigated, since only actual data instances are kept in the index which typically leads to a large increase of distinct index keys but merely to a modest increase of actual records.

Navigational Index

To efficiently process more complex queries, where not all content-related query conditions can be directly connected to a single preceding tag, we need an additional element-only directory to test the structural matches for tag sequences or branching path queries as shown in Figure 2.7. Lookups to

sec

eid	docid	pre	post
46	2	2	15
9	2	10	8
171	5	1	20
84	3	1	12

Figure 2.7: Navigational index for XML elements.

this additional, more compact and non-redundant navigational index yield the basis for the structural scores that a candidate may achieve for each matched tag-only condition in addition to the content scores.

As an illustration of the query processing, consider the example twig query `//A[.//B[.//"b"] and .//C[.//"c"]]`. A candidate that contains valid matches for the two extracted tag-term pairs `B:b` and `C:c` fetched through a series of block-scans on the inverted lists for `B:b` and `C:c`, may only obtain an additional static score mass c, if there is a common `A` ancestor that satisfies both the content-related conditions based on their already known pre- and postorder labels. Since structural conditions are defined to yield this static score mass c, the navigational index is exclusively accessed through random lookups using an additional B^+-tree on this table. Section 8.4 investigates on different approaches to judiciously schedule these random accesses for the most promising candidates according to the structural selectivities and their already known content-related scores.

Chapter 3

Relevance Scoring Model

In this chapter, we define our computational model for top-k rank aggregations and provide the theoretical justification for the scoring models we apply throughout the thesis and in particular in the experiments section. The chapter also underpins the algorithmic prerequisites for a correct top-k query processing assuming monotonous score aggregations. We focus on the two prevalent ranking paradigms that evolved over the past 30 years in IR history, namely the traditional Vector Space Model (VSM) with its whole family of the so-called TF·IDF ranking functions versus more distinguished probabilistic IR approaches with the classic Robertson and Sparck-Jones relevance weights and the state-of-the-art Okapi BM25 model being the most prominent representatives. We select two particular instances of these models and present them in full detail for their (complementary) salient properties:

1) The *TF·IDF* model in its most commonly used form with a linear influence of the term frequency (TF) and a logarithmically dampened influence of the inverse document frequency (IDF) for the resulting per-term document scores. This model typically yields *highly skewed* score distributions as the TF component may grow unboundedly, but also provides more efficient retrieval for non-conjunctive, top-k-style query evaluations, however, at the expense of a restrained retrieval quality.

2) The *Okapi BM25* model as a well-established representative of the probabilistic scoring models with a smoothed, non-linear influence of all ranking components in the resulting scores. It typically yields *less skewed* score distributions and superior retrieval quality, however, at the expense of slightly increased retrieval costs.

For more convenient handling of local scores, for example for extracting and comparing compact score distribution histograms (see Section 5.2.3),

we usually normalize these values to the interval $[0, 1]$ either through an additional post-processing step or directly at document indexing time. As most of the proposed scoring models in IR require some kind of corpus-wide term statistics as a basic ingredient, the *efficient indexing and normalization* of these scores and *incremental index updates* pose specific challenges to the indexing strategies applied. We discuss various aspects in indexing time versus query time trade-offs for static and dynamic retrieval systems. We present various extensions for Web IR and XML IR and provide a detailed discussion of BM25 and its efficient usage for indexing, because it also serves as the our (probabilistic) scoring extension for XML data.

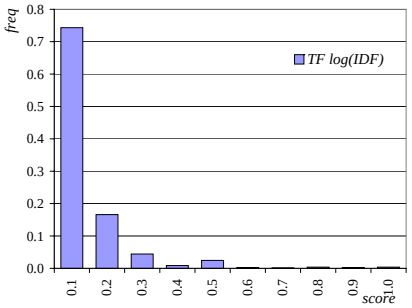


Figure 3.1: Score histogram for the term 'darpa' using a TF·IDF model.

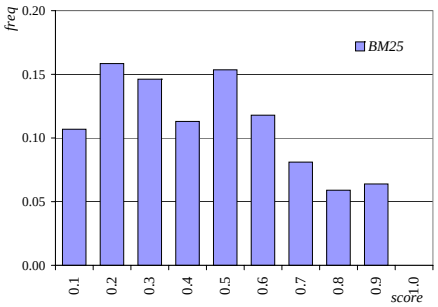


Figure 3.2: Score histogram for the term 'darpa' using a BM25 model.

Figures 3.1 and 3.2 show two score distribution histograms of the inverted list for the rather infrequent term 'darpa' (acronym for *Defense Advanced Research Projects Agency*) with $df_{darpa} = 809$ for the TREC GOV collection with a total of $N = 1,250,000$ documents using a TF·IDF and a BM25 model, respectively, both normalized to the score interval [0,1]. We see that BM25 creates a much more homogeneous distribution than TF·IDF. When combining multiple lists with such a high skew in a non-conjunctive manner, TF·IDF is much more likely to return documents at the top ranks that only qualify through a single match in one list having a high local score that dominates all the remaining query conditions.

Note that we do not examine Statistical Language Models (see, e.g., [LC01, ZL04]) for top-k query processing at this point, although we could principally support these as well, as long as they fall into the common scheme of precomputing local scores and combining these scores using some kind of monotonous score aggregation (e.g., using summation over (log-transformed) basic relevance probabilities). In fact, the 2-Poisson term relevance model and the BM25 weighting function derived thereof (described in Subsection 3.2.1) were formulated before the language modeling idea came

along. However, they may be recast as an elementary form of language model.

3.1 Vector Space Model

Consider a Cartesian product space $D_1 \times \ldots \times D_m$ over domains $D_1, \ldots, D_m$, and a dataset $D \subseteq D_1 \times \ldots \times D_m$ of m-dimensional data points. The data points could be records over structured domains (e.g., product catalog entries) or text documents when the domains capture weights of text terms in a high-dimensional IR feature space. In the latter case, $D \subseteq \Re^m$ or $D \subseteq [0,1]^m$.

Unlike conjunctive queries in a classic DBMS setup, queries on such a dataset are *partial-match queries* in the form of m-tuples $(q_1, \ldots, q_m)$, where $q_i \in D_i$ or $q_i = *$ meaning that we do not care about the ith dimension value. In the text-document IR case, usually a few q_i values are set to 1 (the query keywords) and all others set to 0 (but there are other approaches as well). The results $d \in D$ to a query $q = (q_1, \ldots q_m)$ do not necessarily have to match all non-zero q_i values or have non-zero components d_i; rather we would like to retrieve approximate matches to this condition according to some similarity measure. We assume that for each domain D_i there is a similarity function $s_i : D_i \times D_i \rightarrow [0,1]$. For numerical domains such as year or price, the similarity function could simply be the absolute difference; for structured domains with categorical values such as $Model$ (of cars) or $Genre$ (of movies or books) the similarity function needs to be explicitly defined for all value pairs (recall Section 2.1.2).

For a query $q = (q_1, \ldots q_m)$ and a data instance $d = (d_1, \ldots, d_m)$, the local, domain-specific similarity functions are aggregated into a global similarity function $score : (D_1 \times \ldots \times D_m) \times (D_1 \times \ldots \times D_m) \rightarrow \Re$, with $score(d,q) = aggr\{s_i(d_i, q_i) \mid i = 1..m\}$, where $aggr$ is a *monotonic* aggregation function from $[0,1]^m$ to $\Re$. A widely used aggregation function for this purpose would be summation, thus yielding $score(d,q) = \sum_{i=1}^{m} s_i(d_i, q_i)$; popular alternatives include weighted summation, product (with a probabilistic interpretation), or maximum. The result of a top-k query q then is given by the k data points d for which $score(d,q)$ is among the k highest values of all similarities between q and any other data point d.

Our framework for processing top-k queries is based on sorted access to the data in descending order of similarity scores for each dimension. Especially for processing IR-style index lists, where the lists for frequent text terms may be very long, random lookups into these lists would incur extra disk IOs which are orders of magnitudes more expensive than a sorted access step (with occasional asynchronous and sequential disk reads). The TA fam-

ily of threshold algorithms operate on lists L_i, for $i = 1..m$, which, for a given query value q_i, return data points d_j in descending order of $s_i(d_i, q_i)$, or $s_i(d)$ for short (assuming $q_i = 1$). The implementation uses B^+-tree indexes and scans the inverted index lists in sorted order of scores for individual keys, or looks up keys that exactly match a condition and then merges the results of a forward and a backward scan for neighboring keys (e.g., numerical attribute values with small absolute distance to the query value). When given a query q with specified conditions $q_1, \ldots, q_m$, we assume, without loss of generality, that all dimensions are specified or, equivalently, consider only the subspace of dimensions for which the query specifies values.

3.1.1 TF·IDF Family of Scoring Functions

Table 3.1 shows the family of most commonly used weighting schemes in classic IR [BYRN99]. Recall from Section 2.1.1 that tf_{ij} denotes the *term frequency* of term t_i in document d_j, i.e., the absolute number of occurrences of t_i in d_j; and df_i denotes the *document frequency* of term t_i across the entire corpus, i.e., the absolute number of documents containing t_i. Depending on the specific retrieval application, one may choose between a natural, logarithmically dampened, or augmented and smoothed influence of the TF and IDF components with different effects on retrieval efficiency and result performance. Furthermore, to prevent the score values from growing unboundedly, different normalization approaches can be applied to the resulting document score vectors, such as the L1 or L2 norm, as well as more sophisticated normalization steps that would already incorporate an aggregation step for a given document and query vector such as the Cosine measure (see Section 3.1.2).

Note that the TF·IDF approach in general has often been criticized for being too "ad-doc" and lacking proper mathematical or statistical justification, because it has not explicitly been derived to approximate a specific (postulated) term distribution or a more sophisticated probabilistic relevancy model. However, in practice, these schemes have a long tradition in IR and proved to be effective and work robustly throughout a broad range of IR applications, and some of the smoothed variants indeed are strikingly similar to the well-justified, probabilistic retrieval models (see Section 3.2).

We refer to Equation 3.1 as the most commonly used instance of TF·IDF scoring models that has a natural influence of the TF component and a logarithmically dampened influence of the IDF component and generates non-normalized, potentially unbounded and pseudo-continuous score values.

$$s_i(d_j) \quad = \quad tf_{ij} \cdot \log_2(idf_i)$$

	TF	IDF	Normalization
Natural	tf_{ij}	$\frac{N}{df_i}$	L1-norm: $\frac{1}{\sum_i s_{ij}}$
Logarithmic	$1 + \log(tf_{ij})$	$\log(\frac{N}{df_i})$	L2-norm/Cosine: $\frac{1}{\sqrt{\sum_i s_i(d)^2}}$
Augmented	$0.5 + 0.5\frac{tf_{ij}}{\max_{t_{ij} \in d_j}\{tf_{ij}\}}$	$\log(\frac{N+0.5}{df_i+0.5})$	

Table 3.1: Typical variations of TF and IDF components used in IR applications.

$$= \quad tf_{ij} \cdot \log_2(\frac{N}{df_i}) \tag{3.1}$$

Indexing vs. Query Time Trade-offs

We now refine the basic TF·IDF Formula 3.1 and introduce a simple extension that allows for an efficient *online normalization* of scores at indexing time.

$$\begin{aligned} s_i(d_j) &= \frac{tf_{ij}}{\max_{t_i \in d_j}\{tf_{ij}\}} \cdot \frac{\log_2(\frac{N}{df_i})}{\log_2(N)} \\ &= \frac{tf_{ij}}{\max_{t_i \in d_j}\{tf_{ij}\}} \cdot \log_N(\frac{N}{df_i}) \end{aligned} \tag{3.2}$$

In the experimental setups, we will refer to the above normalized TF·IDF Formula 3.2 as our default TF·IDF model with a natural (but per-document normalized) TF component and the logarithmically dampened (and globally normalized) influence of the IDF component. This slight modification is especially interesting for our top-k framework in two ways:

1) The local TF component *including the normalization step* can be efficiently precomputed in-memory at indexing time whenever a document has been completely parsed and is about to be stored in the inverted lists.

2) Due to the linear influence of the TF component, which is preserved even after the per-document normalization step, this scoring function generates *highly skewed*, Zipf-like score distributions when applied to the whole collection.

The first issue makes the scoring function attractive for indexing very large collections, where indexing performance becomes crucial. With the

above formulation, we can treat the IDF component as a constant for each term during query execution time that does not affect the order of index list entries for a given query term according to the TF component and allows us to provide the combined TF·IDF scores in a *dynamic view*. This view can be generated on demand using order-preserving nested-loop joins, where the inverted lists are sorted in descending order scores for the TF component and the IDF component is dynamically joined by the outer loop for each of the query terms individually. This prevents us from having to rebuild the entire inverted index whenever new documents are added and enables the engine to perform *continuous indexing and querying* with iterative (lazy) updates of the IDF statistics and cheap incremental updates in the inverted lists.

The second issue also depicts a performance factor for query execution times, since a highly skewed distribution already yields the top ranked documents for a multi-keyword query among the top ranks of at least one of the index lists with a high probability, whereas the scores for the major amount of candidates at the remaining ranks of each list quickly converge to 0. Note that by normalizing the IDF part with the global maximum IDF value $\log_2(N)$, we in fact utilize a huge base for the logarithm – namely the collection size N – which makes the TF component the dominating part of this scoring model. We will review this issue in the experiments section and examine these efficiency versus effectiveness trade-offs between different scoring approaches such as TF·IDF and the probabilistic variants; see also [BC05] for a very recent study of indexing versus query time trade-offs in dynamic information retrieval systems.

3.1.2 Vector Space Aggregations

While the precomputation of local scores and the materialization of the inverted index structure is part of the indexer, the efficient aggregation of these local scores for multi-dimensional queries is part of the query processing.

Recall from Section 1.3.2 that for the basic top-k query processing, we maintain for each index list L_i the following information: a current scan position pos_i and the respective current score $high_i := s_i(d)$ for the document d at the current scan position pos_i in L_i. We maintain for each record or document d that was already encountered in at least one of the L_i: a set $E(d)$ of dimensions for which we already computed a score $s_i(d)$, and a partial score $worstscore(d) := \sum_{i \in E(d)} s_i(d)$ and $bestscore(d) := worstscore(d) + \sum_{i \notin E(d)} high_i$ (e.g., using summation as $aggr$). Then way can safely stop query executions and return the top-k documents as soon as

$$\min\{worstscore(d) \mid d \in \textit{top-}k\} \leq \max\{bestscore(d) \mid d \notin \textit{top-}k\} \ .$$

In order to provide a correct algorithm, we need to provide monotonous updates on both the lower and upper bounds for each candidate item seen so far, that is in particular, we may never decrease the lower bound $worstscore(d)$ or increase the upper bound $bestscore(d)$ for any candidate at a later time of the index scans, because that would make the $worstscore(d)$ and $bestscore(d)$ bounds useless and prevent the algorithm from finding the correct point of termination.

Monotonous Score Aggregations

With a few algebraic tricks, we can in fact incorporate most of the score aggregation functions commonly used in IR to support a wide range of IR applications. These include

- *summation* as mentioned already, i.e.,

$$score(d,q) = \sum_{i=1}^{m} s_i(d) \ , \tag{3.3}$$

- *minimum* and *maximum*, i.e.,

$$score(d,q) = \min|\max_{\ i}\{s_i(d)\} \ , \tag{3.4}$$

- the *product* with a summation over log-transformed scores, i.e.,

$$score(d,q) = \prod_{i=1}^{m} s_i(d) = \exp\left(\sum_{i=1}^{m} \log s_i(d)\right) \ , \text{ or} \tag{3.5}$$

- the *Cosine measure* with L2-normalized document and query vectors, i.e.,

$$score(d,q) = \frac{\sum_{i=1}^{m} s_i(d) \cdot q_i}{\sqrt{\sum_{i=1}^{m} s_i(d)^2 \ \sum_{i=1}^{m} q_i^2}} \tag{3.6}$$

$$= \sum_{i=1}^{m} s_i(d) \cdot q_i \quad for \ \left\{ \begin{array}{l} \sqrt{\sum_{i=1}^{m} s_i(d)^2} = 1 \\ \sqrt{\sum_{i=1}^{m} q_i^2} \quad = 1 \end{array} \right. \tag{3.7}$$

One thing that all of these aggregation functions have in common is that they can compensate weak matches (or even local scores of 0) for some query conditions by higher scores at other query dimensions, which is an excellent property for IR and the reason why they are often referred to as *compensation functions* [GBK01]. We will review this issue for non-conjunctive (aka. "andish") query evaluations for text and XML data (see Sections 4.1 and 8.1.3).

Non-Monotonous Score Aggregations

Semantically richer scoring models in IR could also incorporate non-monotonous facets for ranking that would take evidence from the query "as a whole" into account. Note that the underlying scoring function itself does not necessarily have to be monotonous; we merely have to provide *monotonous* $[worstscore, bestscore]$ *bounds* in order to guarantee a correct algorithm, i.e., we could decide to keep more generous bounds for partially evaluated candidates which would in turn render the candidate pruning less effective. Particularly interesting, non-monotonous score aggregation include

- the *Cosine measure* with non-normalized document and query vectors,
- *proximity-based phrase scores* with combined local term weights and phrase distances, or
- *element compactness* for ranking XML subtrees with regard to the amount of nodes spanned by the result element.

While it is immediately evident that most of the current major search engines such as Google or Yahoo apply proximity-aware ranking functions, there are hardly any publications on this issue for text IR [HT95, NO00] or XML IR [AYFSX03]. Most of the existing approaches merely provide an ad-hoc combination of local scores with some notion of minimal phrase distance which is either integrated as part of the result post-processing, or it is even applied during query query executions for early candidate pruning, but in the form of a non-monotonous aggregation which is not suitable for a viable top-k approach. None of the existing work pursues a top-k algorithm as skeleton. The integration of monotonic and non-monotonic scoring issues and their efficient implementation for efficient top-k ranking is an open research issue. We leave this particularly interesting issue for future work.

3.2 Probabilistic Scoring Models

3.2.1 Probabilistic IR

As opposed to the rather ad-hoc TF·IDF family of scoring models, probabilistic IR approaches aim at establishing probabilistically derived scoring models. The TF component is referred to as a relevance weight that reflects the importance of a term in a document; and the IDF component is a specificity weight that refers to the selectivity of a term across the collection. The probabilistic model initially proposed by Robertson and Sparck-Jones

is the beginning of a long series of evolutionary refinements that finally led to the Okapi BM25 model that has meanwhile become prevalent in current benchmark settings such as TREC.

Robertson & Sparck-Jones Weights

The original relevance weighting model [RJ76, Rob81, IDF], referred to as RSJ in the following subsection, aims at a compact and simple approximation of the basic probabilistic relevance model shown in Equations 3.8 and 3.9. Given a corpus for N document out of which R documents are explicitly marked as relevant, the RSJ approach aims to model the probability p_i of a document d containing a term t_i given that d is relevant versus the probability q_i of d containing t_i given that d is not relevant. Robertson and Sparck-Jones show in their initial work from 1976 that under some simple assumptions, the goal of relevance ordering according to those probabilities can be achieved by assigning weights to query terms and scoring a document by aggregating those weights using plain summation.

$$p_i = P(d \textit{ contains } t_i \mid d \textit{ is relevant}) \tag{3.8}$$

$$q_i = P(d \textit{ contains } t_i \mid d \textit{ is not relevant}) \tag{3.9}$$

Then the RSJ weight is

$$w_i = \log \frac{p_i\,(1-q_i)}{(1-p_i)\,q_i} \tag{3.10}$$

With a total of n_i documents containing the term t_i and r_i out of the R documents being relevant, this becomes

$$p_i \approx \frac{r_i}{R} \quad \text{and} \quad q_i \approx \frac{n_i - r_i}{N-R} \tag{3.11}$$

and with smoothed parameter estimation using traditional Lidstone smoothing [Lid20], this yields the classic *Robertson and Sparck-Jones* formula for relevance weighting:

$$w_i = \log \frac{(r_i+0.5)(N-R-n_i+r_i+0.5)}{(R-r_i+0.5)(n_i-r_i+0.5)} \tag{3.12}$$

With very little or no relevance information at all (i.e., $R = r = 0$, see also [RW97]), this is just

$$w_i = \log \frac{N-n_i+0.5}{n_i+0.5} \quad \text{for } R=0 \tag{3.13}$$

$$\tag{3.14}$$

and with n_i being much smaller than N, we have can approximate these weights as

$$w_i \approx \log \frac{N + 0.5}{n_i + 0.5} \qquad \text{for } n_i \ll N \tag{3.15}$$

which already exhibits roughly the same form as the IDF component in current probabilistic models such as BM25 and shows that the ad-hoc notation of the smoothed IDF component in many TF·IDF models is in fact an RSJ weight.

Retrieval with IDF weights alone might work well for relatively small corpora (e.g., a handcrafted digital library system where IR actually originates from) and precise multi-keyword queries, but it would be infeasible for a Google-like 8 billion pages Web corpus and the typical two-term keyword queries, because it does not discriminate documents that indeed contain the same combination of keywords but with different term frequencies.

So far, a profound justification for using IDF values in relevance ranking was given; what has still been lacking at that point was a probabilistic justification for the TF component.

Term Eliteness & 2-Poisson Model

As for the TF component, another probabilistic model is utilized. The central assumption of the so-called *term eliteness model* [RW94] is that each term (or word) represents a concept; and that a given document is either "about" the concept or not, i.e., the term is elite in the document or not. The original terminology, as well as the statistical model described below, has been taken from Bookstein and Swanson (1974) and Harter (1975). Here, eliteness is a latent property that cannot be observed directly. However, if we assume that the text of the document is generated by a simple unigram language model, then the probability of any term being in any given position depends on the eliteness of this term in that document.

If we take all the documents for which this particular term is elite, then we can infer the distribution of within-document frequencies we should observe. If all documents are the same length, then the distribution would be approximately Poisson. If we take instead all the documents for which this term is not elite, we will again see a Poisson distribution (presumably with a smaller mean). As this observation of term eliteness is much more subtle than counting simple term occurrences, eliteness cannot be computed in advance, unless all documents in the corpus would be judged manually for a given query word (or concept); so if we consider the collection as a

whole, we rather assume that we will observe a mixture of two Poisson distributions. This 2-Poisson mixture is the basic Bookstein/Swanson/Harter model [HAR75] for within-document term frequencies.

As for specificity weights, the RSJ relevance weighting model is then reused, thus applying it to query-term eliteness rather than to mere query-term presence or absence.

3.2.2 Okapi BM25

The weighting function today known as Okapi BM25 [RW94] was derived from the previous probabilistic models as an approximation of the 2-Poisson model for term relevance weighting and the RSJ model [RJ76] for term specificity weighting. The final BM25 model was developed as part of the Okapi experimental system out of a series of so-called best match (BM) functions and is probably the best-known and most widely used such extension. It was first applied under its official name at the third TREC benchmark conference [RWHB+95] in 1994 and has meanwhile become the predominant scoring method in the TREC benchmark series.

BM25 first estimates the full eliteness weight from the usual presence-only RSJ weight for the term, then approximates the TF behavior with a single global parameter k_1 that controls the rate of growth for the TF component. Hence, the TF value provides probabilistic evidence of term eliteness and should give partial credit to the document. This credit should rise monotonically from zero if $tf_{ij} = 0$ and approaches the full eliteness weight asymptotically as tf_{ij} increases. In general, the first occurrence of the term gives highest evidence; successive occurrences give successively smaller increases.

Finally, BM25 makes a correction for the document length $length(d_j) = \sum_{i=1}^{m} tf_{ij}$, thus relaxing the equi-document-length assumption of the original 2-Poisson model. On the assumption that one reason for a document to be long is verbosity on the part of the author (which would suggest simple document-length normalization by the TF component), but that a second reason is the topic coverage of the document (which would suggest no normalization), BM25 does partial normalization. The degree of this normalization is controlled by a second global parameter b. In order to make the normalization relatively independent of the units in which document length is measured, it is defined in terms of the average length $avg(length(d))$ of documents in the collection. The resulting combined weights can be expressed as

$$s_i(d) = \frac{(k_1 + 1)\ tf_i}{K + tf_i} \cdot w_i \ , \tag{3.16}$$

where w_i is the usual RSJ weight and

$$K = k_1 \left((1-b) \ + \ b \ \frac{length(d)}{avg(length(d))} \right) \ . \tag{3.17}$$

Then the aggregated score $score(d_j, q)$ of a documuent d_j for a query $q = (qtf_1, \ldots, qtf_m)$ is

$$score(d_j, q) \ = \ \sum_{i=1}^{m} \frac{(k_1+1)\ tf_{ij}}{K + tf_{ij}} \cdot \frac{qtf_i}{k_2 + qtf_i} \cdot \log \left(\frac{N - df_i + 0.5}{df_i + 0.5} \right) \tag{3.18}$$

The combined BM25 model provides a smoothed, non-linear influence of the TF and IDF components. Again, the IDF part is a constant for a given term in a corpus. Note that the original definition of BM25 utilizes an additional ranking component, namely the query term frequency (QTF), that aims to model the relevance of a query term in a given query and is smoothed in a similar way as the TF component, where qtf_i now denotes the frequency of the term t_i in the query. The QTF component is often omitted (i.e., assuming $qtf_i = 1$ and $k_2 = 0$), since most short keyword queries do not exhibit any redundant keywords, anyway. However, it is a corpus-independent constant at query execution time as well, and as such, it does not affect the indexing part.

On-the-fly Normalization for BM25

The resulting per-term BM25 scores themselves are not naturally normalized, but they can easily be normalized online (i.e., at document indexing time without an additional post-processing step) using an upper bound score estimation, thus taking advantage of the strong saturation effect used for the TF, QTF and IDF components. A simple limit analysis for the TF component yields:

$$\lim_{tf_{ij} \to \infty} \frac{(k_1+1)\ tf_{ij}}{K + tf_{ij}} \ = \ 1 \tag{3.19}$$

That is, this expression quickly converges to 1 largely independently of K and k_1 for a reasonable scale of k_1, K and tf_{ij} values and, hence, provides normalized output scores already (with a similar result for the QTF component). Now, for the IDF component, we can perform the following analysis:

$$\lim_{df_i \to 1} \log \left(\frac{N - df_i + 0.5}{df_i + 0.5} \right) \ = \ \log \left(\frac{N - 0.5}{1.5} \right) \approx \log \left(\frac{2}{3} N \right) \tag{3.20}$$

Using the above limits indeed yields *global* (i.e., term independent) maxima for the TF, QTF and IDF components. Using these maxima prevents us from performing an additional post-processing step for global score normalization without crushing a major amount of local scores into very small score intervals, since the overall skew is not too high. The above limit for the IDF component can be further improved by allowing only terms with a minimum df_i value of 10 or more which is a reasonable fit for many large real-world collections and eliminates very rare keywords which mostly occur due to typing mistakes, thus yielding an even tighter upper bound. Very frequent terms (i.e., stop words), on the other hand, are eliminated from indexing as well. Assuming that the most frequent remaining terms have a document frequency of at most $df_i = N/2$, we get a lower bound for the IDF component of

$$\lim_{df_i \to \frac{N}{2}} \log\left(\frac{N - df_i + 0.5}{df_i + 0.5}\right) = \log\left(\frac{N - \frac{N}{2} + 0.5}{\frac{N}{2} + 0.5}\right) \tag{3.21}$$

$$= \log(1) = 0 \ . \tag{3.22}$$

Note that the IDF component is defined in BM25 with the default smoothing parameter of 0.5, such that terms with $df_i > N/2$ are assigned a negative value which is typically adjusted by using a small static and positive score in the inverted lists or by not indexing those terms at all.

3.3 Combined Scoring Models – Web IR

Basic extensions for Web IR incorporate the boosting of term weights according to the HTML tag they occur in, the extraction of anchor texts from external pages, and authority ranking based on link analysis. These techniques are commonly used in Web benchmarks such as the TREC Web track [CH04] and provide some particularly interesting challenges for efficient top-k query processing and indexing. Subsection 3.3.3 provides a novel approach for combining BM25 with these highly skewed authority scores. The resulting combined BM25·PageRank scoring model has been successfully applied at the TREC 2004 Web track and Terabyte track, and the 2005 Terabyte Track Efficiency task [CCS05], respectively (see the TREC experiments in Sections 9.7.4 and 9.7.5).

3.3.1 Multiple Weighted Fields

The exploitation of the HTML structure for term weight adjustment can be considered as the first step toward handling semistructured data. Thus, we

take advantage of the explicit markup (tags) and the structural emphases as given in semistructured – tagged or loosely annotated – documents and yet use a simple text model (bag-of-words) and unstructured queries for retrieval.

Tag	Boost Factor
`<URL>`	×4
`<TITLE>`	×4
`<H1>`	×3
`<H2-H6>`	×2.5
`<TH>`	×2
`<BOLD>`	×2
`<A>`	×1.5
`Other Emph.`	×1.5

Table 3.2: Boosting factors for various HTML tags used.

We count each individual occurrence of t_i in d_j with regard to the weighted field or enclosing tag that the term occurs in, e.g., a term is counted twice when it occurs in a `BOLD` tag, see Table 3.2.There is no gold standard for these weights, and they in fact incur a substantial amount of hand-tuning. Note that as we change the individual TF values, we also have to adjust the document length $length(d_j) = \sum_i tf_{ij}$ and the resulting document weight vector, e.g., for using the BM25 formula over the modified corpus statistics [RZT04].

3.3.2 Link Structure & Anchor Texts

Another common technique from Web IR is the usage of *link anchor texts*, i.e., text contents contained within an HTML `A` tag from external Web pages, to adjust the term frequencies of the actual page that is referenced to by such a link. In particular, anchor text has been found to provide a significant boost to the quality of results for *named page* finding or *home page* finding tasks [EM03] as specified, for example, in the TREC 2003 Web track [CH04] that aims to simulate these types of query tasks on a closed Web corpus. This technique even allows to identify relevant documents, that do not actually contain the query term themselves, but might have been "annotated" by the author of another page with a few keywords that the author most probably has intentionally picked as good descriptors of the link's target page. Thus, these annotations inherently exhibit a high similarity to actual user queries looking for a particular home page or known site.

Then the term frequency is the sum of the local term frequencies plus the number of occurrences of that term in an external link anchor, either counted

as the plain number of occurrences or weighted by a respective boost factor for the `A` tag, see Table 3.2. Note that this technique obviously is very sustainable to link-spam, but investigating on this issue would go beyond the scope of the present work, also because the Web corpora used in the experiments in fact depict a "clean" and widely trustworthy excerpt of the real Web.

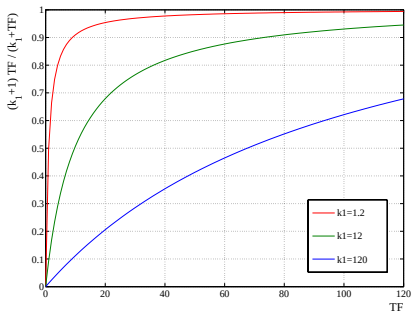
3.3.3 Global Document Weights

Link analysis and authority ranking algorithms such as PageRank [BP98] and HITS [Kle99] have become an indispensable technique for ranking Web documents that provide a query-independent relevance criterion based on the authority of a Web page in the link graph. A common problem is that these algorithms typically generate very skewed score distributions with very low average values compared to the basic BM25 scores. PageRank even models a probability distribution over visitation probabilities over all nodes in the link graph, where all PageRank values add up to 1; and HITS uses an iterative L2 normalization step over authority and hub vectors, with a similar effect on the resulting scores. Moreover, multiplying scores from two skewed distributions (for example BM25 and the original PageRank values) would yield a distribution with an even higher skew which would not be beneficial for retrieval quality. In particular, the highly skewed PageRank component would dominate the resulting scoring model.

Combining PageRank and BM25

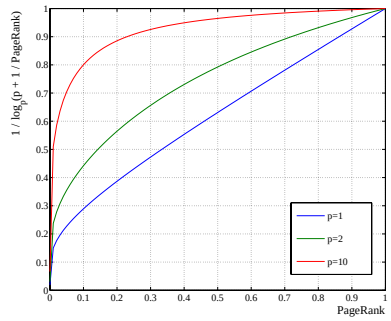
Taking these considerations into account, we can actively design a new scoring model with precisely the desired properties. We therefore take a closer look at the smoothing functionality of the BM25 components as shown in Figure 3.3.

It turns out that we can emulate a similarly concavely shaped and normalized curve for the otherwise very small PageRank values over the interval $[0, 1]$ using a composition of reciprocal values and logarithmic smoothing for the PageRank values $r(d_j)$ with a large logarithmic base p as denoted in Figure 3.4. Then this log-smoothed PageRank component can be directly multiplied into the per-term BM25 scoring model simply by exploiting the distributive law. The resulting combined scores can then be query-independently stored in the inverted lists. This way, all precomputed per-term scores $s_i(d_j)$ for a document d_j are adjusted equally according to its global PageRank $r(d_j)$. Conceptually, the new PageRank component is treated like one of the original BM25 components of similar scale whose influence in the ranking



$$score_{BM25}(d_j, t_i) \cong \frac{(k_1+1)\ tf_{ij}}{K+tf_{ij}}$$

Figure 3.3: BM25-style smoothing of the TF component.



$$score_{PR}(d_j) = \frac{1}{log_p\left(p+\frac{1}{r(d_j)}\right)}$$

Figure 3.4: Log-smoothing of the extended PageRank component.

can be further tuned by the new p parameter:

$$\begin{aligned} score(d_j, q) &= score_{PR}(d_j) \cdot score_{BM25}(d_j, q) \\ &= \sum_{i=1}^{m} \frac{1}{log_p\left(p+\frac{1}{r(d_j)}\right)} \cdot \frac{(k_1+1)\ tf_{ij}}{K+tf_{ij}} \cdot \log\left(\frac{N-df_i+0.5}{df_i+0.5}\right) \end{aligned} \tag{3.23}$$

The Effect of Concavity for Scoring

Figure 3.5 displays the resulting combined BM25·PageRank model as a function of the TF and PageRank values for some fixed DF and QTF and the tuning parameters k_1, k_2, b, and p being set to 1.2, 1.2, 0.75, and 12, respectively. The new scoring function is *concave* in all input parameters, namely in the TF and DF components, the query term frequency QTF (which is not displayed here due to the lack of plotting dimensions), as well as the smoothed PageRank values. The model quickly saturates for large values of TF, QTF, DF and PageRank. Small differences at low input values, for example a jump from a TF value of 2 to 3, yield a stronger effect on the resulting document score than larger differences at higher input values, for example compared to a jump from 10 to 15 or more, and, thus, the BM25 model yields significantly less skewed score distributions in the inverted lists and is less sustainable to synthetically created spam pages with some huge TF values than a non-smoothed TF·IDF model could be. In fact, concavity and fast saturation seems to be an axiomatic demand for successful IR sco-

ring models [FZ05], with the potential to model user-perceived relevance in much better way.

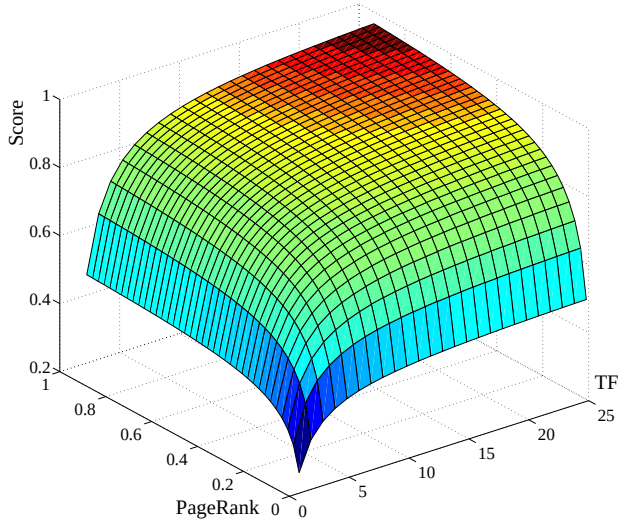


Figure 3.5: Combined scoring model using BM25 per-term scores and global PageRank values.

Our approach diverges from (partly very recent) related work [CRZT05] using a weighted sum of BM25 scores and logarithmically smoothed and (smallest-to-1) normalized PageRank values. We believe that this novel approach is superior to the existing techniques for top-k query processing in two ways:

1) Top-k query processing is further performed on fully precomputed and materialized inverted term lists, where each list directly corresponds to the combined relevance ranking score for a local content-related term score, i.e., a TF·IDF or BM25 model, and authority scores derived from PageRank or HITS. By multiplying the global document scores directly into the inverted term lists, we avoid separate sequential scans or random accesses to an additional, highly selective inverted list for the global document scores. The p parameter that now controls the base of the logarithm adopts the role of the weights in a weighted sum and determines the influence of the PageRank value onto the combined scores.

2) Conceptually, our approach is equivalent to first normalizing the PageRank such that the smallest value is mapped to 1, then multiplying the

local term scores with the log-smoothed global document scores, and afterwards renormalizing the resulting combined scores back to 1. Given the above formalization 3.23, we merge these three steps into a single iteration over the inverted lists. This way, the required indexing time and space is significantly improved, since the proposed combined scoring model directly creates normalized scores as output and essentially saves two rounds of normalization iterations (the latter for the whole inverted index) which makes indexing a factor of two more efficient and does not require any additional temporary index structures.

Note that in contrast to the IDF and QTF components, PageRank values have to be combined with the local TF components of BM25 and be materialized upon index creation to support efficient sorted access, because they are not a constant at query runtime and certainly do affect not only the final document ranking but also the order of all the local index list entries.

3.4 Scoring Models for Semistructured Data

Good scoring models in XML IR are an active research issue. In [TSW05b], we introduced a novel and comprehensive framework for scoring and combining structural and content-related query conditions for XML data with efficient top-k-style query processing.

Content term conditions in structured queries can be evaluated both in conjunctive mode or in "andish" mode. In the first case, all terms must be found, but nethertheless different matches yield different scores. In the second case, a node matches a content condition of the form `/"`t_1 t_2 `..."`, if its content contains at least one occurrence of at least one of the terms t_1, t_2, etc. It matches the full-content condition `.//"`t_1 t_2 `..."`, if its full-content contains at least one occurrence of at least one of the search terms. In the first case, the significance (e.g., derived from frequencies) of a matched term influences the score and the final ranking, but documents (or subtrees) that do not contain a specified term at all are dismissed.

3.4.1 Content Scores

For content scoring we make use of statistical measures that view the content or full-content of a node n with tag A as a bag of words:

1) the *full-content term frequency*, $ftf(t, n)$, of term t in node n which is the number of occurrences of t in the full-content of n;

2) the *tag frequency*, N_A, of tag A which is the number of nodes with tag A in the entire corpus;

3) the *element frequency*, $ef_A(t)$, of term t with regard to tag A which is the number of nodes with tag name A that contain t in their full-contents in the entire corpus.

Now consider a content condition of the form `A//"`$t_1 \ldots t_m$`"`, where `A` is a tag name and t_1 through t_m are terms that should occur in the full-contents of a subtree. The score of node n with tag `A` for such a content condition should reflect:

- a monotonic aggregation of the ftf values of the terms t_1 through t_m (or tf values if we use the child rather than the descendant axis), thus reflecting the *relevance* of the terms for the node's content,
- the *specificity* of the search terms, with regard to $ef_A(t_i)$ and N_A statistics for all node tags, and
- the *compactness* of the subtree rooted at n that contains the search terms in its full-content.

In the following, we will focus on the NEXI-style descendant axis (i.e., the full-content case) as the much more important case for XML IR with vague search; the case for the child axis follows analogously. Our scoring of node n with regard to condition `A//"`$t_1 \ \ldots t_m$`"` uses formulas of the following type:

$$score(n, //A[t_1 \ldots t_m]) \quad = \quad \frac{\sum_{i=1}^{m} relevance_i \cdot specificity_i}{compactness(n)} \ ,$$

where relevance$_i$ reflects ftf values, specificity$_i$ is derived from N_A and $ef_A(t_i)$ values, and compactness considers the subtree or element size for length normalization. Note that specificity is made XML-specific by considering combined tag-term frequency statistics rather than global term statistics, only. It serves to assign different weights to the individual tag-term pairs with regard to the specificity of a term for a given element type which translates the role of the IDF component – a common technique from probabilistic IR – to the XML case.

We could now specialize this formula into a simple TF·IDF-style measure, but an important lesson from text IR is that the influence of the term frequency and element frequency values should be sublinearly dampened to avoid a bias for short elements with a high term frequency of a few rare terms. Likewise, the instantiation of compactness in the above formula should also use a dampened form of element size. To address these considerations, we

have adopted the popular and empirically usually much superior Okapi BM25 scoring model (originating in probabilistic IR for text documents [RW94], see also Section 3.2.2) to our XML setting, thus leading to the following scoring function:

$$score(n, //A[t_1 \ldots t_m]) = \tag{3.24}$$

$$= \sum_{i=1}^{m} \frac{(k_1+1)\ ftf(t_i,n)}{K + ftf(t_i,n)} \cdot \log\left(\frac{N_A - ef_A(t_i) + 0.5}{ef_A(t_i) + 0.5}\right) \tag{3.25}$$

with

$$K = k_1\left((1-b) + b\frac{\sum_{t\in\ full\ content\ of\ n} ftf(t,n)}{avg\{\sum_{t'} ftf(t',n') \mid n'\ with\ tag\ A\}}\right) \tag{3.26}$$

Note that the function includes the tunable parameters k_1 and b just like the original BM25 model. The modified Okapi function provides a dampened influence of the ftf and ef parts, as well as a compactness-based normalization that takes the average compactness of each element type into account. Figure 3.6 shows a simple hill-climbing optimization of the k_1 parameter for the modified BM25 formula using the INEX 2004 benchmark setting which shows a significant increase of the Mean-Average-Precision (MAP) value (see Section 9.6) from 0.171 to 0.187 for $k_1 \approx 10.5$ compared to the Okapi default value of 1.2. Contrariwise, a variation of the of the b parameter did not exhibit any improvement compared to the default value of $b = 0.75$.

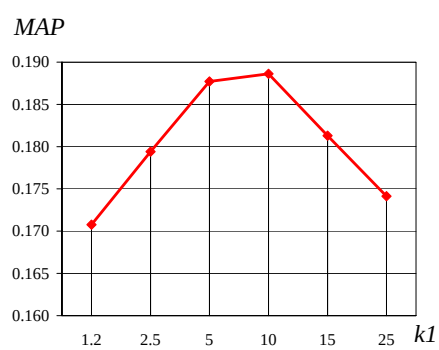


Figure 3.6: Hill-climbing optimization of the k_1 parameter of the extended BM25 model.

This leads to the final element-specific parameterization of the extended BM25 model as indicated in Figure 3.3 for the element types `article`, `sec`,

`par`, and `fig` for the INEX collection [INE]. It was applied similarly to all XML collections in the experiments section with the parameters k_1 and b set to 10.5 and 0.75, respectively, for all element types. With regard to overall retrieval quality, the above formula would also allow a more elaborated parameter optimization for individual element types which would go beyond the scope of this thesis.

Tag	N	$avg(length(n))$	k_1	b
`article`	12,223	2,903	10.5	0.75
`sec`	96,709	413	10.5	0.75
`par`	1,024,907	32	10.5	0.75
`fig`	109,230	13	10.5	0.75

Table 3.3: Element-specific parameterization of the extended BM25 model.

3.4.2 Structural Scores

In non-conjunctive retrieval, a result document (or subtree) should still satisfy most structural constraints, but we may tolerate that some tag names or path conditions are not matched. This is useful when queries are posed without much information about the possible and typical tags and paths, e.g., when the XML corpus is a federation of datasets with highly diverse schemas. In the INEX benchmark, a collection of IEEE CS conference and journal publications, the situation arises because of the large number of different tags and the user's a-priori ignorance about certain content terms occurring in sections or subsections or paragraphs or captions, etc.

Our scoring model essentially counts the number of navigational conditions (or tag-only conditions) o_j that are still to be satisfied by a result candidate d and assigns a small and constant score mass c for every condition that is matched. This structural score mass is combined with the content scores and aggregated with each candidate's $[worstscore(d), bestscore(d)]$ interval. In our setup we have set $c = 1$, whereas content scores were normalized to $[0, 1]$, i.e., we emphasize the structural parts. Note that it is still important to identify non-satisfiable conditions as early and efficiently as possible, because this can reduce the *bestscore* of a result candidate and make it eligible for pruning.

The overall score of a document or subtree for a content-and-structure (CAS) query is the sum of its local scores at all the matched structural and navigational query conditions. For content-only (CO) queries, i.e., mere keyword queries, the document score is the sum, over all terms, of the maximum

per-term element scores for the same document. If TopX is configured to return entire documents as query results, the score of a document is the maximal score of any subgraph in the document.

3.4.3 Common Framework

Note that the way we precompute the content scores for each tag-term pair, with regard to individual corpus statistics for each element type that is identified by the tag name, treats each of these element types as a separate collection of elements with their full-contents. Thus, evaluating a query pattern of the type `A["a"]` in a strict notion, i.e., with a separate inverted list for each of these tag-term patterns, corresponds to the original, document-based BM25 model over the collection of all elements of type `A`.

In particular, we do not mix local scores originating from different BM25 models for different element types for the final result ranking. Multidimensional content conditions that refer to the same element type, e.g., for the target element of type `A` in the query `A["`a_1`, ..., `a_m`"]`, are evaluated for each element of type `A` individually, and all result elements are ranked after their aggregated scores, thus referring to the very same type of score aggregation that the original BM25 formula would perform. For multi-element path queries, we simply aggregate the partial scores for different query subconditions that refer to different element types, e.g., for the support element `A` and the target element `B` in the query `A["`a_1`,...,`a_{m_1}`"]//B["`b_1`,...,`b_{m_2}`"]`, where `A` and `B` address two distinct BM25 models. Then all the previous assumptions for probabilistic IR and BM25 (see Sections 3.2.1 and 3.2.2) remain valid for the XML case, too.

Also, TopX could easily support other scoring models as long as they are monotonous and fall into the framework for relevance, specificity, and compactness. Furthermore, our extended BM25 approach may also be combined with other scoring components such as global document weights, e.g., using PageRank values also for linked XML collections. Good XML scoring functions, e.g., based on statistical language models, are an active research issue, in particular in the INEX benchmark series [INE]; our BM25-based scoring model achieved a very good result quality on the INEX benchmark 2005 (see Section 9.8.4), where our runs for the strict content-and-structure (SSCAS) task ranked at positions 1 and 2 among all submissions [TS05].

XML Scoring Example

As an example, consider the three documents shown in Figure 3.7, and the following twig query:

```
//A[.//"a" and .//B[.//"b"] and .//C[.//"c"]]
```

The query uses the self-or-descendant axis, so the scoring refers to the full-contents of elements. Within document d_1 the twig pattern is matched only once by the elements $(2, 4, 5)$; in document d_2 there are three matches $(1, 6, 7)$, $(1, 3, 5)$, $(1, 4, 5)$; document d_3 contains matches for the individual conditions but does not satisfy all path conditions (there is no B element among the descendants of either one of the two A elements). Assume that the ef values of all three tag-term pairs A:a, B:b, and C:c are the same. Then the best one among all matching triplets is $(1, 4, 5)$ within document d_2, which has a global score of $2/9 + 2/3 + 1$, thus making d_2 the top result.

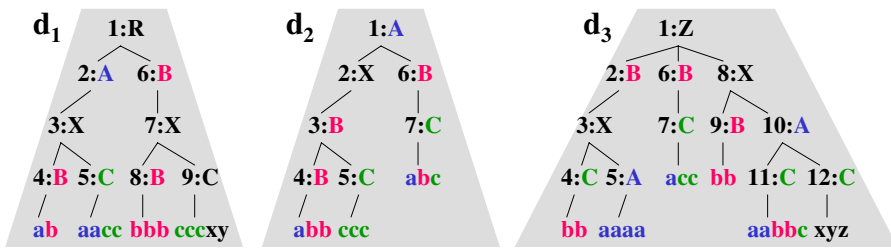


Figure 3.7: XML example documents.

3.5 Query Term Weights & Boosting Factors

The dynamic weighting of query terms, e.g., for dynamically modeling the IDF or QTF components, or for expressing term similarities in query expansions or relevance feedback environment, calls for a modified score aggregation function, e.g., in the form of a weighted sum with $score(d, t_i) = \alpha_i \; s_i(d)$.

For some applications, e.g., the "+" operator of a keyword query, simply scaling the local scores by a term weight α_i may be not enough to emphasize a term in a document *independently* of its local score $s_i(d)$. In addition, we instruct the index scans to prioritize index lists for mandatory terms by "boosting" their per-term scores by an additional constant value β_i that dominates the local scores $s_i(d)$ and disables the compensation property of the score aggregation at the boosted query dimension. This leads to the following modified score aggregation function

$$score(d, t_i) \;=\; \begin{cases} \alpha_i \, (\beta_i + s_i(d)) & for \; d \in L_i \\ \\ 0 & otherwise \end{cases} , \qquad (3.27)$$

where the $\alpha_i \in [0,1]$ coefficients are either 1 or set to the search-task-specific similarity values, and the $\beta_i \in \Re_0^+$ coefficients are either 0 for optional terms or set to a "high value" for mandatory terms. Note that we can combine different similarity factors that influence the query term weights, such as term similarities, relevance feedback, etc., into a single α_i value that is used in the modified score aggregation.

Also note that this notion of a "high value" for β_i is basically IR-driven, $\beta_i > m$ would safely enforce the presence of that term in the results but might endanger recall; $\beta_i = 1$ most probably suffices already with the option of still being able to compensate that term for a number of very good other matches. In any case, all α_i and β_i values are constants at query execution time, and therefore keep the aggregation function monotonous in the input scores $s_i(d)$. Query terms are only boosted for a candidate item d, if this item is actually present in the respective index list L_i, i.e., if $d \in L_i$; local scores $s_i(d)$ with a value of 0 are not boosted, of course. The modified score aggregation is applied to both worst- and bestscore bounds during query processing which may "stretch" these score intervals.

Introducing query weights and boosting factors may lead to increased scan depths on the corresponding index lists, if an otherwise promising result candidate has very low scores in these lists but needs to be tested for the presence of the mandatory terms. This is exactly the case where random accesses for term-presence testing make sense, and our cost-based random-access scheduler automatically identifies lists with a high potential for candidate pruning and gives higher priority to these lookups in such cases (see Chapter 6).

3.5.1 Relevance Feedback

Relevance feedback [Roc71, RJ76] is an important way to enhance retrieval quality by integrating explicit relevance information provided by a user. In XML retrieval, existing feedback engines usually generate an expanded keyword query from the content of elements marked as relevant or non-relevant. In order to support relevance feedback in XML queries, we extend the NEXI syntax with additional weights for each content constraint similar to the `ftcontains` operator in the XPath 2.0 Full-Text extension (see Subsection 2.2.2). A typical extended NEXI query then looks like the following:

```
//article[about(.,‘‘0.8*XML’’)//*[about(//p,‘‘0.4*IR -0.2*index’’)]
```

The extended NEXI syntax offers both a human readable and machine processable way to incorporate relevance feedback in a structure-aware retrieval engine.

3.5.2 Negative Query Weights

Relevance feedback – and in particular *negative query term weights* – pose an additional challenge to a top-k query processor. With negative α_i coefficients and arbitrary boost factors β_i, we have to generalize the score bounds as follows to maintain monotonous score updates:

$$\begin{aligned} worstscore(d) &= \sum_{i \in E(d)} \alpha_i(\beta_i + s_i(d)) \\ &+ \sum_{i \notin E(d)} \min\{\alpha_i(\beta_i + high_i),\ 0\} \qquad (3.28) \\ bestscore(d) &= \sum_{i \in E(d)} \alpha_i(\beta_i + s_i(d)) \\ &+ \sum_{i \notin E(d)} \max\{0\ , \alpha_i(\beta_i + high_i)\} \qquad (3.29) \end{aligned}$$

Note that for negative α_i, we are conceptually scanning the inverted lists in inverse order, namely in *ascending* order of $\alpha_i \cdot high_i$ values of local scores. Hence, the best match of a document at a condition with $\alpha_i < 0$ is a local score $s_i(d) = 0$, i.e., if the document does not contain the negated term at all. The modified bounds are more conservative than before, as we have to keep larger intervals for the case that the actual best possible local score for a negated dimension – namely 0 – is reached only at the end of the index scans and no longer at the beginning.

For a detailed evaluation of the TopX engine for structure- and content-aware relevance feedback in XML IR, see our most recent work [ST06b, ST06a, TS05] which takes advantage of the versatile querying options of the engine, including structured queries and negative query weights. Our approach that is inspired by text-based IR basically extends the probabilistic Robertson and Sparck-Jones weighting scheme (see Subsection 3.2.1) to automatically construct a content-and-structure (CAS) query from a content-only (CO) query through user-provided relevance feedback.

Chapter 4

TopX Core Query Processor

The TopX core query processor is responsible for the top-k and candidate bookkeeping. The algorithmic skeleton is based on Fagin's work on the Combined Algorithm (CA), using a round-robin-like – but multi-threaded – sorted access scheduling baseline. As for random access scheduling, a simple cost model is utilized that aims at a balanced amount of sorted and random accesses according to the c_R/c_S ratio between sorted and random access costs. Here, the exact c_R/c_S ration is considered a system-dependent parameter that can easily be empirically derived through an initial test query. As opposed to the original CA algorithm, TopX uses our *Last-Probing* approach for RA scheduling by default that splits the SA and RA scheduling in two strictly separated phases, thus saving all RAs until to the end of the query processing which already results in major amount of access costs saved and significantly improved query runtimes compared to CA (see Section 6.4.1 for details on the scheduling). For the current chapter, however, we may just assume that a static scheduling strategy for sorted access (e.g., round-robin) and random access (e.g., one random lookup every c_R/c_S sorted accesses for the currently best candidate in the queue) is given.

The TopX core algorithm is extended by an expensive predicate random access scheduler, coined *Min-Probing*, to be able to also resolve more complex query predicates that could not at all – or only with very high costs – be resolved through sorted access alone. The Min-Probing approach also allows for random access scheduling to auxiliary data structures other than the inverted lists, such as as an additional term-to-position index for phrase matching.

A modified scoring aggregation function with a combination of boosting weights for individual query conditions and adaptive thresholds allows to implement not only either strict conjunctive or "andish" query evaluations, but also a mixed conjunctive (but ranked) and andish retrieval mode, with some

query conditions being marked as mandatory and others as optional. Moreover, the core TopX query processor already provides an efficient and versatile algorithmic baseline for exact top-k query evaluations that outperforms existing approaches without necessarily employing any heavy probabilistic machinery.

4.1 Conjunctive vs. Andish Query Evaluations

Unlike for traditional, conjunctive database joins, the result quality in IR applications greatly benefits from non-conjunctive query evaluations, where the final result ranking for multidimensional queries is primarily determined through the aggregation of local scores for each individual query condition, and low-scoring conditions (including 0-scores for some conditions) can be compensated by a result object, if it otherwise exhibits some extraordinarily high local scores. Note that the general result output in this mode is "almost" conjunctive, as the top-scored items typically need to have a good score among most query conditions to qualify for the top ranks, but some conditions can be relaxed on-the-fly, when not all conditions can be matched among the items in the collection. Hence, this mode is often coined *"andish"*. Therefore, an axiomatic demand [FZ05] for good scoring models in IR seems to be a controlled skew over the local score distributions, with a dampened and sublinear influence of all input parameters.

Note that andish is not the same as a disjunctive query evaluation in database jargon, it would rather have to take all 2^m subsets of m query conditions into account for retrieving valid partial results and ranking them by a subsequent (partial) sort algorithm to return the top-k results. This would be extremely expensive to emulate in a default query language such as SQL, with multiple outer joins on all attributes, thus forcing full scans on all input lists. We will review this issue also for "andish" XML IR as opposed to Boolean XPath (see Section 8.1.3).

A top-k-style query processor, on the other hand, is inherently "tuned" for andish query evaluations, as partial results are combined (or accumulated) in-memory for each candidate object individually, and a score-based threshold condition is iteratively tested for algorithm termination.

4.1.1 Mixed Mandatory & Optional Query Conditions

For ranked retrieval with some or all conditions being marked as mandatory, the scores for these mandatory query terms should still reflect the relevance of the term for a given element, i.e., our precomputed TF·IDF- or BM25-based

content scores. Yet an overly strict Boolean interpretation of the + operator would make us run into the danger of losing recall at the lower ranks.

We therefore employ the slightly modified score aggregation of the form $score(d, q) = \sum_{i=1}^{m} \alpha_i(\beta_i + s_i(d)$ from Section 3.5. Recall that $s_i(d)$ is the original per-term score, α_i are query term weights, and β_i is set to 1 if the term is marked as mandatory (+) and 0 otherwise. Note that these β_i are constants at query evaluation time, and since the modified scores are taken into account for both the $worstscore(d)$ and $bestscore(d)$ bounds of all candidates, the boosting factors "naturally" enforce deeper sequential scans on the inverted index lists for the mandatory query conditions, typically until the final top-ranked results are discovered in those lists. Still, weak matches for the remaining non-boosted query conditions may be compensated by a result candidate through high-scored matches in the mandatory query conditions.

Note that mandatory search conditions could now be cast into a notion of expensive text predicates, too, with random access probing (see the next section), but the way the scores are defined allows a significant contribution of the precomputed local scores $s_i(d)$ onto the final aggregated score and affects both the worst- and bestscore bounds. We believe the latter option is more elegant for top-k query processing, because it does exploit the local scores $s_i(d)$ and does not necessarily lead to additional random accesses.

4.1.2 Adaptive Min-k Thresholds

If we take a closer look at the modified score aggregation function, we see that $\sum_i \alpha_i \, \beta_i$ for all i with $d \in L_i$ is a *static* score threshold that merely depends on d's presence in the respective inverted lists and that is independent of d's actual local scores $s_i(d)$.

$$score(d, q) = \sum_{i=1}^{m} \alpha_i \left(s_i(d) + \beta_i\right) \tag{4.1}$$

$$= \underbrace{\sum_{i=1}^{m} \alpha_i \beta_i}_{\text{static score threshold}} + \underbrace{\sum_{i=1}^{m} \alpha_i s_i(d)}_{\text{local term scores}} \tag{4.2}$$

Then we set the initial *min-k* threshold to this static, query-dependent value $\sum_{i=1}^{m} \alpha_i \beta_i$ that *all* top-k results have to overcome. Again, the exact choice of the β_i value is an IR-style trade-off between effectiveness and recall; $\beta_i \geq m$ would for sure enforce all mandatory query conditions to be matched by the top-k results but might make us run into the danger of losing a

substantial amount of recall at the lower ranks; $\beta_i = 1$ most probably yields the desired boosting effect already.

4.2 Expensive Predicates

The use of auxiliary query hints in the form of expensive text predicates such as phrases ("), mandatory terms (+), and negation (−) can significantly improve the retrieval results of an IR system. The challenge for a top-k based query processor lies in the *efficient* implementation of these additional query constraints and their adaptation into the sorted versus random access scheduling paradigm. Generalizing the notion of expensive predicates as provided in [CwH02], we obtain the following definition:

Definition 4.2.1 (Expensive Predicate) *An query predicate is called an expensive predicate, if it cannot at all – or only through very high cost – be resolved through sorted access alone.*

From the above definition, it follows directly that, for example, phrase tests are expensive, because phrases cannot be tested with a standard inverted index at all; and negations are expensive, because for a strict negation test, we would have to scan entire lists regardless of the document's score in the negated condition(s). Associating a predicate condition with one or more query conditions that inevitably entails increased index access cost calls for a smart approach to minimize the amount of expensive random access probes that are about to be scheduled for yet untested predicate conditions.

4.2.1 Random Access Scheduling for Expensive Predicates

As for the final score that a candidate is assigned with a mixture of "normal" (unrestrained) query conditions and expensive predicates, we consider a combination of local scores and a static (i.e., known) predicate gain that a candidate additionally accumulates if it matches the predicate condition. As a special case, this gain may be just the local scores themselves at the predicate condition, when predicates are used as binary filters, e.g., for binary phrase matching.

On the one hand, it is clear that some documents must be fully probed (for every predicate in the query), which include at least the top-k answers in order to determine their query scores and ranks in the query results. On the other hand, since k is usually small, only some answers are requested,

and complete probing for every document is not necessary. To avoid this prohibitive cost, our goal is to stop as early as possible for *each* candidate object. In fact, the major amount of objects may not need to be probed at all, if they can never be among the top answers. Following [CwH02], this calls for the following definition:

Definition 4.2.2 (Necessary Predicate Probe) *Consider a ranking query with scoring function aggr and retrieval size k. A probe for a predicate $i \in P(d)$ for a candidate object d is necessary, if the top-k answers with respect to aggr cannot be determined by any algorithm without performing the probe, regardless of the results of other probes.*

The incremental testing and scheduling of expensive predicate probes asynchronously of the inverted list lookups requires us to extend the candidate's data structure as initially described in Section 1.3.2 by an additional bit vector

- $P(d)$, i.e., a set of unevaluated predicate dimensions that d still has to match.

$P(d)$ is defined by the query structure and initially contains the same subset $P(d) \subseteq \{1, \ldots, m\}$ of predicate dimensions for all candidates encountered during the inverted list scans. For example, in the query

undersea "fiber optics cable" -satellite

the initial $P(d)$ field for each candidate contains dimensions for "fiber", "optics", "cable" (both because of the phrase), and "satellite" (for its negation), i.e., $P(d) = \{2, 3, 4, 5\}$.

We define the score-gap $gap(d)$ that a candidate accumulates, if all predicate conditions in $P(d)$ are matched as

$$gap(d) \;=\; \sum_{i=1}^{m} s_i(d) \quad for\ i \in E(d) \cap P(d)\ . \tag{4.3}$$

In order to keep updates for a candidate's score bounds monotonous, we need to modify the lower $wortscore(d)$ bound to deal with the additional uncertainty factor induced by the $P(d)$ field:

$$worstscore(d) \;=\; \sum_{i \in E(d) \cap \bar{P}(d)} s_i(d) \tag{4.4}$$

$$bestscore(d) \;=\; \sum_{i \in E(d)} s_i(d) \;+\; \sum_{i \in \bar{E}(d)} high_i \tag{4.5}$$

$$\tag{4.6}$$

This way, we are more restrictive on the worstscore bound, thus assuming that, although we might have seen the candidate in the inverted lists already, the predicate, e.g., a phrase, might not be matched in the end. The bestscore bound, on the other hand, remains unchanged, thus assuming that, though we have not tested a predicate yet, we can be sure that d will not accumulate *more* score mass than before. Recall that we have to keep both the worstscore and bestscore updates monotonous for each candidate and at each step of the query processing.

Now we are in a position to define a Min-Probe scheduling condition: Schedule RAs for all $i \in P(d)$, only if

$$worstscore(d) + gap(d) \quad > \quad min\text{-}k \tag{4.7}$$

which is a direct translation of the above definition of necessary predicate probes for our algorithmic paradigm with a focus on inexpensive sorted access.

Note that the value of $gap(d)$ increases when we gain additional information about the candidate in $i \in E(d)$ and $worstscore(d)$ would increase, such that the scheduling decision tends to be reached only at a *late* stage of the query processing for most candidates. That is, we schedule expensive random lookups for the unresolved predicates on d, only if we know in advance that this will promote the candidate to the (intermediate) top-k results and in turn will lead to an increase of the $min\text{-}k$ threshold (which might lead to an increased pruning of remaining candidates). This way, only the most promising candidates are tested; for the great majority of candidates, $worstscore(d) + gap(d)$ will never exceed $min\text{-}k$.

Further note that sequences of random accesses to test multiple predicates may be interrupted as soon as $bestscore(d) \leq min\text{-}k$, i.e., the candidate fails sufficiently many predicate conditions and is dropped from the queue. Very lowly selective predicates (e.g., very infrequent phrases), however, may lead to an increased amount of predicate tests and degrade the performance of our algorithm.

4.2.2 Negation

The semantics of negations for a non-conjunctive, i.e., "andish", query processor is all but trivial. To cite the authors of the NEXI specification [TS04a], "the user would be surprised if a '−' word is found among the retrieved results". This leaves some space for interpretation and most commonly leads to the conclusion that the negated term should merely lead to a certain score penalty; yet we do not want to completely eliminate all documents containing one of the negated terms as in a conjunctive setup. Hence, a match to

a negated query condition does not necessarily render the result irrelevant, if good matches to other content-related query conditions are detected, and we would run into the danger of losing a substantial amount of recall by strictly enforcing negations.

Therefore, in contrast to mandatory search conditions, the scoring of negated terms is defined to be *constant* and *independent* of the term's actual content score $s_i(d)$. Similarly to the structural query constraints introduced in the XML scoring model (see Section 3.4.2), a result candidate merely accumulates some additional static score mass if it does *not* match the negated term. Let $N(d) \subseteq P(d) \subseteq \{1, \ldots, m\}$ be the set of query conditions marked by a '$-$', then the aggregated score of a candidate item d is defined as

$$score(d, t_i) = \begin{cases} \alpha_i \left(s_i(d) + \beta_i\right) & for\ d \in L_i \\ \alpha_i \beta_i & for\ d \in N(d) \wedge d \notin L_i \\ 0 & otherwise \end{cases} \tag{4.8}$$

with $\beta_i = 1$ for $i \in M$ or $i \in N$. Note that we need at least one non-negated query condition as basis to perform sequential scans on.

This quickly leads us back to the notion of expensive predicates and the minimal probing approach. A random lookup for the candidate to the inverted list L_i for the negated conditions $i \in N(d)$ is scheduled, before the document gets promoted into the top-k results after a successful negation test, i.e., if it does not contain the negated term and accumulated the static score for the unmatched negation. In the current setup, this static score mass has been set to the same value $\beta_i = 1$ that was provided for the usual term boosting as well as the structural query constraints in the XML case; the $\alpha_i \in [0, 1]$ coefficients furthermore allow for a weighting of these negations.

4.2.3 Phrase Matching

For phrase matching, we store all term offsets in an auxiliary database table. Again, phrases are interpreted as expensive predicates and term proximities for the inverted index entries are solely tested by random accesses to the offset table, using the minimal probing approach already described for the Min-Probe scheduling. The only difference now is to determine whether a candidate may aggregate the additional score mass provided for the phrase-related conditions into its overall $worstscore(d)$ that is then used to determine its rank among the top-k results. In order to keep these score aggregations monotonous in the precomputed content scores, phrase lookups are treated as *binary filters* in the current implementation only.

Phrase Negations

Similarly to the single-term negations, phrase negations are defined to yield a static score mass c for each candidate that does not contain the negated phrase. Single-term occurrences of the negated phrase terms are allowed, though, and do not contribute to the final element score unless they are also contained in the remaining query. If one inverted list that is part of a phrase has been entirely scanned sequentially, we may safely reset the worst- and bestscore of all candidates that have not yet been fully evaluated in the related query dimensions of that phrase.

4.2.4 Frequent Terms

Frequent terms with a very high selectivity (i.e., document frequency) can be considered as soft-filters, too. They yield long inverted lists, typically with low slopes in their local scores. Scheduling sorted access for those lists would be very expensive and would lead to unnecessarily high sequential IO costs. Figure 4.1 shows a highly skewed distribution of the index list lengths of all

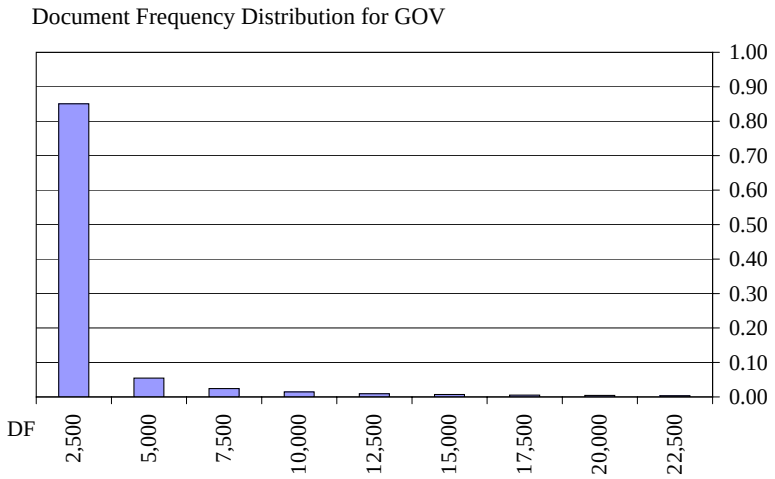


Figure 4.1: Distribution of index lists lengths for the GOV collection with a total of $N = 1,250,000$ documents (with stop words removed).

index terms in the TREC GOV collection (see the experiments overview in Section 9.2.1), i.e., more than 98 percent of all search terms occur in less than 1 percent of the documents (even with stop words and numbers being removed). Note that this observation is purely selectivity-driven, it is not affected by the scoring model applied.

As an example, consider the two short keyword queries

"trec nist" vs. *"trec home"*.

Both refer to the homepage of the TREC benchmark website and should return the link to `http://trec.nist.gov` among the top-ranked results (Google aggrees on this). The term "trec" has a document frequency of $df_{trec} = 109$, "nist" has a document frequency of $df_{nist} = 19,517$, and "home" finally has a document frequency of $df_{home} = 285,633$ out of 1,250,000 documents in the corpus.

Then sorted access for "trec" would fetch the whole list in a single batch of sequential disk IO, so the runtime cost for that term is negligible. Then the final ranking would be fought out among the remaining index lists for "nist" or "home", respectively. Now the rank of TREC home page in local list for "home" could be almost an arbitrary position among the 285,633 entries. So sorted access for the significantly longer "home" list would generally not be beneficial for early stopping, given that we are looking for a small amount of top-ranked pages, only. In fact, many short keyword queries are composed of a mixture of highly specific (infrequent) terms and one or two rather unspecific (frequent) terms.

Now, the frequent terms are by definition very highly selective, and the expensive predicate probes for them will not often fail. So probing a candidate would only be necessary when some of the less selective term conditions have already been matched with a high score through sorted access. That is, we can in fact treat frequent terms with a document frequency of more than about 2 percent (which corresponds to the rightmost histogram cell for $DF = 22,500$ in Figure 4.1) as soft filters in the notion of an expensive predicate and solely probe them through random accesses.

4.3 Multi-threaded Top-k Query Processing

The TopX core processor performs parallel sequential index scans with large prefetch batches and individual index list buffers that are continuously filled by an additional tier of low-level buffer threads. Sorted accesses are performed on top of these list buffers by a separate thread for each index list, whereas candidate pruning and scheduling decisions are performed iteratively by the TopX main thread, i.e., after a whole batch b of sorted access step when all scans threads are suspended. At these synchronization points, all threads responsible for sorted accesses are paused and the shared data structures are maintained.

Figures 4.3 and 4.2 demonstrate the advantages of the multi-threading architecture in two small experiments, both conducting a batch of 50 TREC 2003 Webtrack queries for the topic distillation task on the GOV collection

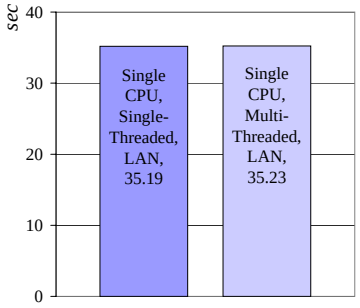


Figure 4.2: Multi-threading vs. single-threading on a single CPU system.

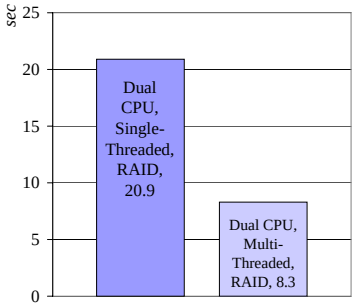


Figure 4.3: Multi-threading vs. single-threading on a dual CPU system.

(see also [CHWW03] and Section 9.2.1), but on two different hardware configurations. Figure 4.2 shows a wallclock runtime of 35.2 seconds for both the multi- and single threaded configuration (the latter scheduling SA batches in a simple round-robin style) using a single-CPU notebook (with a 1.6 Ghz Centrino CPU) connected to the Oracle server via a 1 Gigabit LAN. This demonstrates the I/O boundedness of the algorithm for this particular configuration which is exactly what we would expect. The situation changes, however, when queries are executed directly on the server machine that also hosts the Oracle database (with a 3Ghz dual Xeon CPU) and tuples are read directly read from the RAID disks. Figure 4.3 shows that the wallclock runtime significantly drops from 20.9 seconds in single-threaded mode to 8.3 seconds in multi-threaded mode which demonstrates that a single thread cannot exhaust the full I/O bandwidth on the server. So multi-threading is a crucial performance issue, in particular on multi-CPU machines.

The general TopX architecture comprises a three-tier, multi-threaded hierarchy consisting of

1) the *main thread* that iteratively maintains the data structure for candidate bookkeeping and optionally updates probabilistic predictors for candidate pruning and adaptive scheduling decisions after a batch of b index accesses,

2) the *scan threads* that continuously read and join input tuples from the inverted list buffers for a batch of $b_i \leq b$ sorted accesses, and

3) the *buffer threads* that continuously refill a small buffer cache and controll the actual disk I/O for each inverted list involved in a multi-dimensional query.

This three-level architecture builds on the observation that candidate pruning

and scheduling decisions are expensive and should be done only iteratively, and joining and evaluating score bounds for candidate may incur high CPU load (in particular for path query evaluations), while the actual sequential index accesses are not critical in terms of CPU load.

This way, the scan threads are totally decoupled from each other. Synchronization (object locking) for shared data structures only takes place when a candidate is pulled from the cache and the queue is updated, or when (occasionally) a candidate is found to be promoted into the top-k queue which happens much less frequently than updates on the candidate queue (following the same argumentation why the expensive predicate scheduling remains efficient). Note that the lightweight *min-k* threshold and predicate gap tests for expensive predicate probing may be employed for each candidate update, i.e., after each sorted access step, directly by the respective scan thread.

The top-k operator itself allows for a pipelined and (almost) non-blocking implementation modulo some static prefetching time needed to initialize the candidate queue. Then any nested top-k operator can be seamlessly integrated into multi-threaded architecture of the TopX query processor by making the scan thread at dimension i the main thread of a nested top-k operator; see Section 7.5 for a detailed description of nested top-k operators.

Futhermore, we divide the task of candidate bookkeeping into two separate data structures: the *cache* and the *candidate queue*.

Cache

The cache is the brain of the TopX query processor. It contains a superset of the candidates currently being contained in the top-k and candidate queue. Splitting the task of candidate management into two separate data structures, namely a hash-based cache and a continuously sorted queue, arises from the observation that only a small subset of the best candidates (according to bestscores) actually has to be kept in the queue at-a-time for testing the *min-k* threshold condition. Hashing has the great advantage of constant lookup and update costs of $O(1)$; and the queue updates, that are accounted by $O(\log q)$ each, are significantly accelerated for a bounded queue of size q. However, to have a correct algorithm, we need the cache to "remember" much more partially evaluated objects than are actually necessary for maintaining the threshold condition with a bounded queue of size q.

Candidate Queue

Various queuing options are conceivable for efficient candidate management. The approaches we investigate range from maintaining up to $2^m - 1$ for all pos-

sible remainder sets $\bar{E}(d)$ over query dimensions in $\{1, \ldots, m\}$ toward using only a plain, unsorted candidate pool. Among our most effective approaches is the aforementioned combination of a bounded queue with a hash-based cache (coined the *Prob-smart* in Section 5.3). Albeit a heuristic, the queue bound q may be chosen in the order of the batch size b which is typically a safe choice (see Section 5.3). Then testing the top candidate in the queue with the kth ranked top-k item allows for a lightweight, any-time *min-k* threshold test for algorithm termination.

4.3.1 Main Thread

Algorithm 3 shows pseudo code for the main routine of the TopX core query processor. It synchronizes the scan threads and notifies a semaphore object that the scan threads are waiting for. It optionally invokes more sophisticated candidate statistics and advanced scheduling decisions.

Queue Garbage Collection

Following the basic top-k query processing introduced in Section 1.3.2, we may safely drop a candidate d from the queue if

$$bestscore(d) \;\leq\; min\text{-}k \;, \tag{4.9}$$

i.e., when d cannot qualify for the top-k results any more. Then, $bestscore(d)$ is iteratively updated for all candidates in the queue by the main thread for pruning, taking the current $high_i$ values at the current scan positions into account.

Note that we also need to consider a "pseudo candidate" $\widetilde{d}$ with $best-score(\widetilde{d}) = \sum_{i=1}^{m} high_i$ (see also Section 5.3), that is conceptually put in the queue, too, prior to invoking the queue garbage collector. $bestscore(\widetilde{d})$ is considered as an upper bound for all yet unseen candidates, in order to prevent the algorithm from stopping too early, namely before any yet unseen candidate d' with $bestscore(d') \leq bestscore(\widetilde{d})$ that might still qualify for the top-k is encountered.

This rather conservative deterministic *min-k* threshold test may be extended by a more aggressive *probabilistic threshold test*, thus yielding an approximative top-k algorithm with great runtime gains, as described in Section 5.

Cache Garbage Collection

TopX may optionally perform an iterative cache garbage collection, too, when main memory consumption becomes crucial (e.g., for large query expansion

runs). Analogously to the queue garbage collection, we may safely drop a candidate d from the cache, if

$$\sum_i high_i \leq min\text{-}k \wedge bestscore(d) \leq min\text{-}k \ , \tag{4.10}$$

because then d may be rediscovered in a subsequent sorted scan but will be pruned immediately, because then $bestscore(d) \leq \sum_i high_i \leq min\text{-}k$. Note that cache garbage collection may be a costly operation as the cache potentially contains much more candidates than the (bounded) queue, and many items, that have only been seen at an early stage of the query processing, are never seen again and hence are not updated by the current $high_i$ values and could remain "stuck" in the cache with a high bestscore.

Basic Random Access Scheduling

Basic random access scheduling in TopX follows the initial work done by Fagin on the Combined Algorithm (CA) and incurs a very lightweight cost model that can be implemented just by counting the number of sorted and random accesses done so far. The basic random access scheduler follows the invariant $c_R \cdot \#RA \leq c_S \cdot \#SA$, i.e., the random accesses scheduler would limit the amount of RAs to one RA every c_R/c_S SAs, thus eliminating the currently best $c_R/c_S \cdot b$ candidates from the queue after a batch of b sorted accesses.

Now, anticipating the results from our scheduling efforts at this point, we generally find a strict separation of the these two scheduling phases to be much more cost-beneficial than the iterative intermixing of SA and RA batches as originally specified by CA. Then our approach, coined Last-Probing in Section 6.4.1, switches from a strict SA mode to RAs-only, at the point when the overall (expected) query costs, denoted as $\#SA + c_R/c_S \cdot \#RA$, is minimized. Using the number of unresolved query conditions $|\bar{E}(d)|$ of candidates d remaining in the queue as an *upper bound* for the number of expected remaining RAs, this minimum is reached, when

$$\#SA = c_R/c_S \cdot \sum_{d \in Q} |\bar{E}(d)| \ . \tag{4.11}$$

This simple extension already outperforms the CA baseline by a large margin, because RAs are typically scheduled at a point when we have gained much more information about the candidate ranks. Note that more sophisticated, probabilistic cost models would also take advanced index lists statistics such as score distribution histograms or even index lists correlations into account

and may also diverge from the strict round-robin SA scheduling baseline as described in Chapter 6.

Note that the algorithm does not require all top-k results to be fully evaluated at all query dimensions to safely return the top-k list. The remaining scores for the k top results can easily be clarified for the final ranking by a few random lookups when the algorithm terminates.

4.3.2 Scan Threads

Algorithm 4 shows pseudo code for the loop that each scan thread processes to handle the next sorted access until it gets terminated by the main thread or reaches the end of the inverted list. Candidate handling is strictly pipelined and parallelized, while all accesses to shared data structures such as the queue and the cache have to be implemented thread-safe. Each thread has exclusive sorted access to an inverted list, with object locking only being done for cache and queue updates. The number of threads may correspond to the number of query conditions or it may be limited to the best $m' \leq m$ lists according to the SA scheduler. All scheduling decisions can be updated iteratively and adaptively.

Sorted Access Batching

Sorted access are batched in blocks of b overall sequential index steps for all content conditions of an m-dimensional query. This block is divided into $b_1, \ldots, b_m$ individual batch sizes with $\sum_{i=1}^{m} b_i = b$ and worked through by an individual thread that exclusively reads input tuples from the inverted list for each individual query condition. After working through the batch b_i steps, each thread suspends its work, synchronizes onto a shared semaphore object, and waits for the notification signal to that semaphore by the main thread. In the meantime, when some threads with short b_i batches are suspended, the CPU load may be shared by other threads with higher batch loads b_i.

In a basic round-robin-style of sorted access scheduling, these b_i batches are of equal size, i.e.,

$$b_i = \frac{b}{m'} \ , \tag{4.12}$$

where $m' \leq m$ is the set of currently active, not yet completely scanned lists. For a more sophisticated SA scheduling, the b_i batches may diverge from round-robin, e.g., taking the different skew of local score distributions into account. In particular, for a CPU-bounded execution environment of the query engine, some of the b_i batches may become temporarily 0 to limit the

number of threads concurrently being active to the m' lists with the highest score gradients, e.g., with m' confirming to the number of CPUs available. These SA scheduling decisions are iteratively redetermined after each synchronization step in a continuously adaptive query optimization manner (see Section 6.3). This enables us to perform hash-joins and score evaluations for different data objects in a truly parallel manner, in particular for the non-negligible computational overhead of complex XPath evaluations.

Note that the actual disk read operations (which are typically not CPU critical) are decoupled from this synchronization architecture by an additional tier of *buffer threads* that constantly aim to refill a small read cache for each index list that the actual scan threads are operating on.

4.3.3 Buffer Threads

As for the third and lowest tier in the TopX thread hierarchy, we distinguish the goal to optimize individual query runtimes versus optimizing query costs in terms of the absolute number of index lists accesses. For the first, we may allow a bit more disk I/O than absolutely required for the actual query evaluation; for the latter we would rather want to stop reading from disk as early as possible.

To optimize query execution time, we need to ensure smooth and continuous, *asynchronous disk operations* throughout the whole query processing. With the above strategy of dividing index scans and garbage collections into different threads, disk operations might get temporarily interrupted at the synchronization points, namely when all scan threads are suspended and the garbage collectors and candidate pruning is active at the main thread. Therefore, apart from the result set prefetching at the database connector (e.g., ODBC or JDBC) or disk caching effects (which we cannot easily control), we add an additional small buffer for each physically stored index list that does not exceed the default batch size that is initially scheduled for the first round of round-robin-like index list accesses (e.g., a maximum of 1,000 tuples). We add an additional tier of threading responsible for the actual disk reads and buffered index lists lookups to completely decouple the physical I/O performance from the query processing. Figure 4.4 depicts this multi-tier buffering architecture.

Then all scan threads solely work on top of these buffers which are constantly refilled by the tier of decoupled buffer threads with asynchronous disk I/O until the query processing terminates. This way, we experience no startup delays after notifying the scan thread which makes multi-threaded scheduling with different batch sizes per thread feasible, because the disk operations are not interrupted. The actual buffer threads are suspended, too,

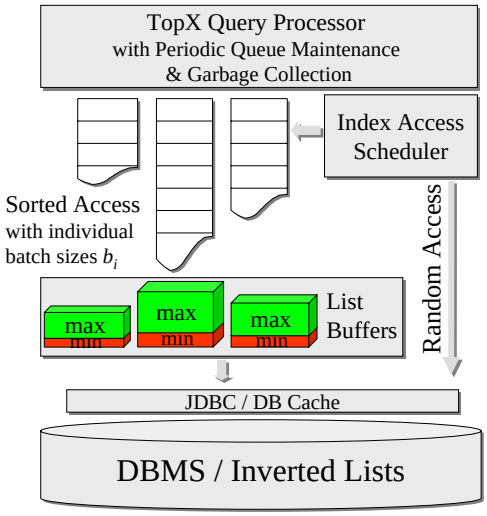


Figure 4.4: Multi-tier buffering architecture of TopX.

when the intermediate read buffer is filled to the maximum value, and they are notified when the buffer falls below some minimum fill threshold (e.g., a minimum of 100 tuples). The maximum buffer sizes may be chosen proportionally to the scheduled index access batch sizes b_i, when the scheduler increasingly strives away from the initial round-robin scheme.

Note that random accesses as triggered by the main thread that directly accesses the inverted lists. This type of access greatly benefits from the internal caching strategy of the underlying DBMS.

4.3.4 Threshold & Continuous Queries

Returning *exactly* the top-k results per query may not always be very helpful in an interactive user session. Often, a user would require to click and browse through various result pages and "explore" a certain topic of interest to satisfy the information need. More general top-k application include *threshold queries*, where all result items need to exceed an initial, user-defined score threshold, and *continuous queries*, where the user may iteratively click through a number of result pages each of which contains k ranked items.

Threshold Queries

Threshold queries are a trivial extension to a top-k query processor. Instead of specifying a value for k, we simply determine a static score threshold Δ to

initialize *min-k* analogously to the boosting threshold described in Subsection 4.1.2. Note that it is also possible to incrementally (and monotonically) increase Δ, if too many intermediate results are found, and thus stop earlier.

Continuous Queries and Top-k Shifts

Continuous queries, however, are a bit more tricky but also much more assistant than a threshold query from a user perspective. These queries benefit from the fact that a user typically first digests an initial result page (e.g., with k being in the order of 10 or 20) and then may decide to further browse for the next page, thus looking at the $k' > k$ results, then again digests that page, and so on. That is, the query processor does not need to precompute the whole result set for a large value of k'; and moreover, the final value of k' is not decisive. Then the overall query cost (and runtime) to retrieve the $k' = s \cdot k$ is divided into s iterative *top-k shifts*. Thus, these top-k shifts are nothing but dynamic threshold queries, where we iteratively reset *min-k'* to 0 and refill the extended top-k' list with the best intermediate k' results.

Algorithm 5 shows pseudo code for this shift operation. The algorithm iterates over all items in the cache and puts those items d with $worstscore(d) >$ *min-k'* into the extended top-k' list, and otherwise puts items with $best-score(d) >$ *min-k'* back into the queue. Note that *min-k'* does not have a fixed value and putting new items into the top-k' list increases *min-k'*, too. The new value of *min-k'* must be lower or equal to the previous *min-k* value that led to the last algorithm termination. Since *min-k'* $\leq$ *min-k*, we may need to resume the scan threads and perform additional index scans until the new *min-k'* threshold condition holds. Note that this requires the cache garbage collector to be disabled, however, since we may not finally prune items from the cache for an unknown value of *min-k'*. Then it is not necessary to restart the whole query, but rather suffices to merely wake up the suspended scan threads that can seamlessly continue their work. Typically, any subsequent top-k shift incurs significantly less execution cost than retrieving the initial k items.

Algorithm 3 Main thread of the TopX query processor.

```
1:  TopXMainThread(Index Lists L_i, Query q=t_i,...,t_m, Batches b_1,...,b_m, Predictor threshold ε)
2:  top-k := ∅;
3:  candidates := ∅;
4:  cache := ∅;
5:  min-k := max(0, Σ_{i=1}^{m} α_i β_i);
6:  activeThreads = 0;
7:  suspendedThreads = 0;
8:  // Start background threads
9:  for all Index Lists L_i (i=1..m) do
10:     threads[i].processIndexList(L_i, b_i);
11:     activeThreads++;
12: end for
13: // Main thread loop
14: while activeThreads > 0 do
15:     // Periodically synchronize with scan threads and iterate top-k workflow
16:     while suspendedThreads < activeThreads do
17:         // Main thread synchronization on semaphore object
18:         semaphore.waitForNotification();
19:         suspendedThreads := #threads waiting for notification;
20:         activeThreads := #threads still being active;
21:     end while
22:     // Update pseudo candidate's threshold
23:     sumHigh := Σ_{i=1}^{m} high_i;
24:     // Queue garbage collection & probabilistic pruning
25:     for all d ∈ candidates do
26:         bestscore(d) := worstscore(d) + Σ_{ν∈Ē(d)} high_i;
27:         if bestscore(d) ≤ min-k | p(d) ≤ ε then
28:             Drop d from candidates;
29:         else
30:             Consider RA on all L_i with i ∉ E(d) according to modified cost-model;
31:         end if
32:     end for
33:     // min-k threshold termination
34:     if topk.size() == k & (candiates == ∅ | bestscore(candidates.top()) ≤ min-k) then
35:         break;
36:     end if
37:     // Cache garbage collection
38:     if sumHigh ≤ min-k then
39:         cache.gc();
40:     end if
41:     // Schedule next round of sorted access batches
42:     scheduler.updateBatches();
43:     // Notify scan threads
44:     activeThreads = 0;
45:     for all Index Lists L_i (i=1..m) do
46:         threads[i].notify();
47:         activeThreads++;
48:     end for
49: end while
50: // Schedule remaining random lookups for final top-k ranking
51: for all d ∈ top-k do
52:     for all i ∈ Ē(d) do
53:         L_i.getRandomScore(d);
54:     end for
55: end for
56: return top-k;
```

Algorithm 4 TopX scan thread.

```
1:  processIndexList(IndexList L_i, Batch Size b_i)
2:  isAlive_i = true;
3:  isSuspended_i = false;
4:  pos_i = 0;
5:  while isAlive & L_i.hasNext() do
6:    // Perform next sorted access to L_i
7:    <docid, score> := L_i.getNext();
8:    d := cache.getCachedItem(docid);
9:    s_i(d) := score;
10:   E(d) := E(d) ∪ {i};
11:   high_i := score;
12:   pos_i++;
13:   // Update worst- and bestscore bounds
14:   worstscore(d) := Σ_{i∈E(d)} α_i(β_i+s_i(d));
15:   bestscore(d) := worstscore(d) + Σ_{ν∈Ē(d)} α_ν(β_ν + high_ν);
16:   // Check expensive predicates (Min-Probing)
17:   gap(d) = Σ_{i∈E(d)∩P(d)} s_i(d);
18:   if worstscore(d) + gap(d) > min-k then
19:     for all i ∈ Ē(d) ∩ P(d) do
20:       checkPredicates(d, i);
21:     end for
22:   end if
23:   // Update top-k list
24:   if worstscore(d) > min-k then
25:     // Commit top-k update
26:     d' := top-k.removeMinkItem();
27:     top-k.insert(d);
28:     candidates.remove(d);
29:     // Adaptive min-k threshold
30:     min-k := max(top-k.getMinkScore(), min-k);
31:     // Former top-k item may still be a valid candidate
32:     if bestscore(d') > min-k then
33:       candidates.insert(d');
34:     end if
35:   else
36:     // Cache & queuing test
37:     if bestscore(d) > min-k then
38:       candidates.update(d);
39:       cache.update(d);
40:     else
41:       candidates.remove(d);
42:     end if
43:   end if
44:   // Suspend & wait for main thread notification
45:   if pos_i mod b_i == 0 then
46:     isSuspended_i = true;
47:     semaphore.notify();
48:     this.waitForNotification();
49:   end if
50:   isSuspended_i = false;
51: end while
52: isAlive_i = false;
```

Algorithm 5 Top-k' shift.

```
topkShift(New result size k')
// Keep previous top-k results and reset new min-k' threshold
top-k' := top-k;
min-k' := max(0, Σ_{i=1}^{m} α_i β_i);
for all d ∈ cache do
  // Extend top-k' list
  if worstscore(d) > min-k' then
    d' := top-k'.removeMinkItem();
    top-k'.insert(d);
    min-k' := max(top-k'.getMinkScore(), min-k');
    // Former top-k' item may still be a valid candidate
    if bestscore(d') > min-k' then
      candidates.insert(d');
    end if
  else
    // Extend candidate queue
    if bestscore(d) > min-k' then
      candidates.insert(d);
    end if
  end if
end for
// Resume scan threads
for all Index Lists L_i (i=1..m) do
  threads[i].notify();
  activeThreads++;
end for
// Continue as in TopX main thread shown in Algorithm 3
// ..
```

Chapter 5

Probabilistic Candidate Pruning

The TA family of threshold algorithms, and in particular the NRA variant with no random accesses being allowed, is conservative in that it stops scanning index lists only when it is certain that no more top-k results can be found. In particular in IR applications, the final scores among the top-ranked results are often very tight, and typically most of these results are equally relevant from a user perspective.

We believe that the given algorithms for query evaluations are overly reluctant in pruning candidates given that the concept of a top-k query has a heuristic nature, anyway. Hardly any end-user would be interested in looking at exactly the k best matches to a similarity query. Rather the rationale of top-k ranking is that users typically find one or a few relevant and novel data items among the top 10 or 20 results. So there is an inherent and unavoidable risk of missing the truly best results (in the subjective judgment of the user). This in turn justifies relaxing the concept of a top-k query into an approximate notion such that the query processor can occasionally tolerate errors: false positives or false negatives with regard to the top-k.

Our approach is based on predicting the total score of a candidate item for which we know a partial score, e.g., the sum of local scores for one or more elementary conditions, but not the total score for all conditions. In doing this, we avoid the overly conservative worst- and bestscore bounds of the original TA family of threshold algorithms by calculating the probability that the total score exceeds a threshold that would make the item interesting for the top-k result. If this probability is sufficiently low, we drop the data item from the candidate list. The probabilistic prediction of the aggregated score involves computing the convolution of the score distributions of different index lists. To this end, we explore a variety of techniques including aggregate score predictors using histograms, efficiently evaluable Poisson estimations, and convolutions based on moment-generating functions

with generalized Chernoff-Hoeffding bounds for the resulting tail probabilities. For approximate top-k results with a small, probabilistically bounded error, the query processor also has these precomputed score histograms or parameterized predictors available and maintains further state information for estimating the score at a list position and the list position for a given score: $score_i : position \rightarrow score$ and $pos_i : score \rightarrow position$ can be derived as two deterministic, reversible functions using histograms or parameterized score distribution estimators.

As the overhead of these techniques is crucial, the details of our bookkeeping and candidate testing strategies are all but straightforward; we explore a wide range of strategies within the paradigm of threshold algorithms based on different setups of priority queues. To the best of our knowledge, our work is the first to present a method for probabilistic top-k queries with a controllable and tunable trade-off between result quality and index access cost. Note that our probabilistic guarantees are not about query runtimes but about query result quality; runtime bounds that hold with high probability have been derived in [Fag99]. Also, our approach should not be confused with probabilistic methods for deriving local and global scores, e.g., probabilistic IR techniques as discussed in Section 3.2.1; we can handle a wide variety of scoring functions as building blocks but our notion of probabilistic guarantees developed in this chapter refers to the approximation of the top-k retrieved data item in a completely scored and exactly ranked result set. Our experiments on various text and semistructured data collections demonstrate an impressive amount of evaluation costs saved, with up to two orders of magnitude compared to the non-approximative baselines and a very good precision versus runtime ratio.

5.1 Top-k Query Processing with Probabilistic Guarantees

Recall from Section 1.3.2 that our family of threshold algorithms is based on the invariants

$$worstscore(d) = \sum_{i \in E(d)} s_i(d) \tag{5.1}$$

$$bestscore(d) = worstscore(d) + \sum_{i \notin E(d)} high_i \tag{5.2}$$

$$\tag{5.3}$$

Suppose we already have k items that are the preliminary top-k results

of a given query q for an arbitray snapshot of the query processing, and let $min\text{-}k := \min\{worstscore(d) \mid d \in top\text{-}k\}$. Then we can prune documents and remainders of index lists for documents whose upper bound cannot exceed the $min\text{-}k$ threshold, i.e., a document d can be dismissed from the candidate queue if $bestscore(d) \leq min\text{-}k$. In this case we say that the threshold test fails.

This consideration is often unnecessarily conservative, because the *expected* remainder score of a document is much lower than the sum of the $high_i$ bounds for $i \notin E(d)$. In particular, with no random accesses being allowed on the index structures (e.g., using the NRA baseline algorithm), exclusive sequential scanning typically makes the $[worstscore(d), bestscore(d)]$ bounds converge only slowly, such that often a small number of good candidates close to the top-k matches is kept in the queue over a long period of the query processing. Figure 5.1 depicts this situation for candidate item d: although d's worstscore never exceeds the $min\text{-}k$ threshold, it has to be kept in the queue for a long time, namely until to the point when d's bestscore falls below $min\text{-}k$ for the first time. Since k is usually small, in the order of 10 or 20, compared to the inverted lists, this in fact happens for the major amount of candidates.

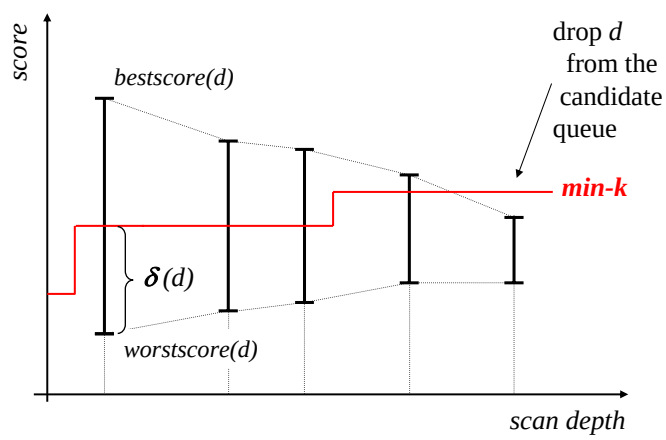


Figure 5.1: Evolution of a candidate's score bounds.

Of course, using plain expectations for pruning would not give us any guarantees for not missing any of the true top-k results. But we would expect that the sum of the $s_i(d)$ scores in the remainder set $\bar{E}(d)$ is lower than the sum of the $high_i$ bounds with very *high probability*. So we are interested in estimating the probability that a document d that we encounter at position

pos_i in the index list L_i, and for which $\bar{E}(d) \neq \emptyset$ holds, qualifies for the top-k results as

$$p(d) \;:=\; P\left[\sum_{i \in E(d)} s_i(d) + \sum_{i \in \bar{E}(d)} S_i > min\text{-}k | S_i \leq high_i \; for \; i \in \bar{E}(d)\right]$$

where S_i denotes the random variable that captures the probabilistic event that document d has a score of $s_i(d)$ in dimension i. With $\delta(d) := min\text{-}k - worstscore(d)$, this is equivalent to

$$p(d) \;:=\; P\left[\sum_{i \in \bar{E}(d)} S_i > \delta(d) \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right] \tag{5.4}$$

Note that when we compute $p(d)$ during query execution we know upper bounds $high_i$ for the unknown scores, thus considering conditional probabilities as denoted above. If the probability $p(d)$ was below some threshold ε (e.g., between 1 and 10 percent) then we might decide to disregard d, without computing its full score, thus introducing a notion of approximate top-k query processing with probabilistic guarantees for the result precision. We refer to condition 5.4 as the *probabilistic threshold test*.

In the following section, we are developing the details for estimating the probability $p(d)$ that a candidate document d with non-empty remainder set $\bar{E}(d) \subset \{1..m\}$ may qualify for the top-k results. The way how we estimate $p(d)$ depends on the assumptions that we make about the distribution of the unknown scores $s_i(d)$ that d would obtain. The following subsections discuss various cases that are of interest from both a fundamental insight and application viewpoint. We will concentrate on the most important case of using summation for score aggregation, and will discuss generalizations at the end of this chapter. Note that summation is the standard choice in IR keyword query processing, with TF·IDF-style scores or probabilistic weights (e.g., Okapi BM25) being precomputed and stored in the inverted index lists.

5.1.1 Convolutions

The *convolution* of two score distribution for two random variables X and Y yields the score distribution for the sum $X + Y$ of the two random variables. Besides their application in statistics, convolutions are also an important tool in data processing, in particular in digital signal and image processing.

Consider the sum $U = X + Y$ of two random variables, where X has the probability density function $f_x(x)$ and Y has the probability density function $f_y(y)$, respectively. For *independent* random variables X and Y,

the convolved sum has the probability density $f(u)$ given by the convolution integrals

$$f(u) = \int_{-\infty}^{\infty} f_x(x)\ f_y(u-x)\ dx \tag{5.5}$$

$$= \int_{-\infty}^{\infty} f_y(y)\ f_x(u-y)\ dy\ . \tag{5.6}$$

Now the task is to determine $f(u)$ given $f_x(x)$ and $f_y(y)$. Obviously, the shape of the convolution density function $f(u)$ depends on the assumptions we make for the input density functions $f_x(x)$ and $f_y(y)$. For a number of cases, $f(u)$ can be computed analytically using an efficiently evaluable closed form, thus following [All90]. A few of the most important ones are listed below:

- The convolution of two *Normal distributions* with zero mean $\mu_1 = \mu_2 = 0$ and variances σ_1^2 and σ_2^2 is again a Normal distribution with zero mean μ and variance $\sigma^2 = \sigma_1^2 + \sigma_2^2$.
- The convolution of two χ^2 *distributions* with f_1 and f_2 degrees of freedom is again a χ^2 distribution with $f_1 + f_2$ degrees of freedom.
- The convolution of two *Poisson distributions* with parameters λ_1 and λ_2 is again a Poisson distribution with parameter $\lambda = \lambda_1 + \lambda_2$. This is a particularly nice property which is directly applicable in many IR tasks.
- The convolution of an Exponential and a Normal distribution is approximated by another exponential distribution. If the original exponential distribution is

  $$f(x) = \begin{cases} \frac{e^{-\frac{x}{r}}}{r} & for\ x \geq 0 \\ 0 & for\ x < 0 \end{cases}\ , \tag{5.7}$$

  and the Normal distribution has zero mean μ and variance σ^2, then for $u \gg \sigma$ the probability density of the sum is

  $$f(u) \approx \frac{e^{-\frac{u}{r}+\frac{\sigma^2}{2r^2}}}{\sigma r\sqrt{2\pi}} \tag{5.8}$$

  In a semi-logarithmic diagram, where $\log(f_x(x))$ is plotted versus x and $\log(f(u))$ versus u, the latter lies by the amount $\sigma^2/(2r^2)$ higher than the former, but both are represented by parallel straight lines whose slope is determined by the parameter r.

- The convolution of a Uniform and a Normal distribution results in a quasi-Uniform distribution smeared out at its edges. If the original distribution is Uniform in the region $a \leq x < b$ and vanishes elsewhere, and the Normal distribution has zero mean μ and variance σ^2, then the probability density of the sum is

$$f(u) = \frac{\psi_0 \frac{(u-a)}{\sigma} - \psi_0 \frac{(u-b)}{\sigma}}{b-a} . \tag{5.9}$$

  Here

$$\psi_0(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} \, dt \tag{5.10}$$

  is the distribution function of the standard Normal distribution. For $\sigma \rightarrow 0$, the function $f(u)$ vanishes for $u < a$ and $u > b$ and is equal to $1/(b-a)$ in between. For finite σ, the sharp steps at a and b are rounded off over a width of the order 2σ.

In the area of digital signal or image processing, convolutions are used for the description of the response of linear shift-invariant systems, and are used in many filter operations. One-dimensional discrete convolutions are written as $z(k) = \sum_i x(i) \cdot y(k-i)$ and often abbreviated as $z = x \oplus y$. Convolutions are

- *commutative*, i.e., $x \oplus y = y \oplus x$,
- *associative*, i.e., $x \oplus (y \oplus z) = (x \oplus y) \oplus z$, and
- *distributive*, i.e., $x \cdot (y \oplus z) = (x \cdot y) \oplus (x \cdot z)$.

Figure 5.2 illustrates the approach for a candidate document d_{10} that we encounter during the sequential scans in the inverted list L_1 for the query term t_1 with the score $s_1(d_{10})$. Let's assume we have a 3-dimensional query consisting of the terms t_1,t_2, and t_3, and we did not see d_{10} in any other index list so far; then $\bar{E}(d_{10}) = \{2, 3\}$ and $\delta(d_{10}) = min\text{-}k - s_1(d_{10})$. In order to predict the probability $P[\sum_{i \in \bar{E}(d_{10})} S_i > \delta(d_{10})]$, we simply convolute the two score distributions for d_{10}'s remainder dimensions, captured by the two precomputed histograms for S_2 and S_3 in this case, to derive the probability that the *sum* of d_{10}'s remainder scores may exceed $\delta(d_{10})$. Then $p(d)$ is determined by the convolution score mass for scores beyond $\delta(d)$ (omitting the conditional probabilities at this point).

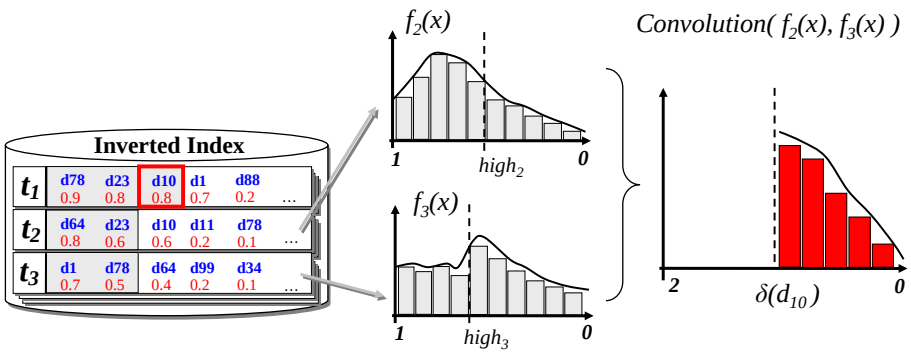


Figure 5.2: Convolution procedure and score prediction for a candidate document with two remainder dimensions.

5.2 Predictors for Aggregated Scores

Among all possible score distributions, the Uniform distribution is the simplest one, but usually not a good fit for real-world score distributions over large text corpora. However, if we do not know any details about the scores, assuming a Uniform distribution is often a convenient and indeed conservative choice. Moreover, computing the convolution of Uniform distributions is still computationally feasible as we will see in Section 5.2.1. The Poisson distribution is a more reasonable fit for realistic score distributions (e.g., using a BM25 scoring model with the default parameterization) and additionally has some nice theoretical properties. As mentioned above, a particularly nice property of the Poisson distribution is that the convolution of m such distributions with parameters $\lambda_1, \ldots, \lambda_m$ is again a Poisson distribution with parameter $\lambda = \lambda_1 + \ldots + \lambda_m$.

Even though parameterized distributions are elegant and efficient to compute, it is often infeasible in practice to capture real score distributions with basic distribution functions and parameter fittings. In such cases, histograms [Ioa03] are commonly used as a compact way to capture arbitrary score distributions.

5.2.1 Chernoff-Hoeffding Bounds for Uniform Distributions

In the absence of any other information, Occam's razor suggests that the simplest assumption about the distribution of unknown partial scores is a *Uniform distribution*. More specifically, we assume that for document d and dimension $i \in \bar{E}(d) = \{1..m\} - E(d)$, where d has not yet been evaluated,

the score $s_i(d)$ is uniformly distributed between $high_i$, the currently known upper bound for the true score, and 0, the assumed lower bound. Instead of 0, we may also use the lowest value that occurs in L_i, provided we have stored this information in the index metadata (i.e., without having to scan L_i to its end). We use continuous distributions rather than discrete ones, as this simplifies the subsequent calculations. We assume that all random variables S_i are independent; feature correlations, or more precisely, the case of *limited feature independence* [Nel95], will be reconsidered in the next step.

Treating each unknown $s_i(d)$ value as a random variable S_i, we thus have to predict the probability $P[\sum_{i=1}^{m} S_i > \delta(d) \mid S_i \leq high_i]$. For two random variables S_1 and S_2 with uniform densities $f_1(x) = 1/high_1$ and $f_2(x) = 1/high_2$ this requires computing the density of the convolution

$$f(x) \quad = \quad \int_0^x f_1(z) f_2(x-z)\ dz. \tag{5.11}$$

Taking into account the fact that each factor is non-zero only within certain intervals, namely, $0 \leq z \leq high_1$ and $0 \leq x - z \leq high_2$, or equivalently $\max(0, x - high_2) \leq z \leq max(x, high_1)$, solving the integral requires the following three cases, assuming $high_1 \leq high_2$ (without loss of generality):

$$f(x) = \begin{cases} \frac{x}{high_1 \cdot high_2} & for \ \ 0 \leq x \leq high_1 \\ \\ \frac{1}{high_2} & for \ high_1 < x \leq high_2 \\ \\ \frac{1}{high_1} + \frac{1}{high_2} - \frac{x}{high_1 \cdot high_2} & for \ high_2 < x \leq high_1 + high_2 \end{cases} \tag{5.12}$$

and a corresponding cumulative distribution in an efficiently evaluable closed form.

Unfortunately, for three and more heterogeneous Uniform distributions, this kind of computation, albeit still simple in principle, leads to a rapidly increasing number of cases regarding integration boundaries that are fairly awkward to handle. In order to avoid these case differentiations, we rather treat the basic density functions for random variables S_i with densities $f_i(x)$ and their convolution $f(x)$ in terms of *moment-generating functions* which is the default strategy in statistics for this case. The moment-generation function for a basic density function $f_i(x)$ then has the form

$$M_i(s) \quad = \quad \int_0^s e^{s \cdot x} f_i(x)\ dx = E[e^{s \cdot S_i}]\ . \tag{5.13}$$

The great advantage of transforming $f_i(x)$ into $M_i(x)$ is that the convolution of moment-generating functions for independent random variables has a moment-generating function of the following very compact form [Nel95]:

$$M(s) \quad = \quad \prod_{i=1}^{m} M_i(s) \ . \tag{5.14}$$

But with Uniform distributions $f_i(x)$ plugged in, this yields a function from which we cannot easily infer the density of the convolution. Instead, we apply *Chernoff-Hoeffding* bounds to the tail probability of the convolution [Nel95, SSS95]:

$$P\left[\sum_{i=1}^{m} S_i > \delta(d)\right] \quad \leq \quad \inf_{s \geq 0} \left\{ e^{-s \cdot \delta(d)} \cdot M(s) \right\} , \tag{5.15}$$

where the infimum on the right-hand side is either the minimum of the Chernoff bound function, computed by finding the roots of the first derivative, or a limit, e.g., for s approaching 0. This computation can be automated using computer algebra tools like Maple and its programming interface OpenMaple.

A great advantage of this approach is that it can be generalized to incorporate distributions other than Uniform ones by simply exchanging the underlying density functions $f_i(x)$ that are used to approximate the actual distributions. Moreover, it can easily handle *heterogeneous* distributions with some scores S_i being uniformly distributed and others following, e.g., Hyperexponential or Zipf distributions. Finally, using results from [SSS95] we can even handle non-independent random variables, although the corresponding *generalized Chernoff-Hoeffding* bounds may not be as strong as in the standard case. Assume that $S_1, \ldots, S_m$ are our random variables of interest. We construct a set of independent random variables $T_1, \ldots, T_m$ such that T_i has the same distribution as the marginal distribution S_i. For a partitioning $\delta = \delta_1 + \ldots + \delta_m$ for the tail quantile of interest consider the Chernoff bounds ε_i with $P[T_i > \delta_i] \leq \varepsilon_i$. [SSS95] have shown that

$$P\left[\sum_{i=1}^{m} S_i > \delta(d)\right] \quad \leq \quad \inf_{\delta_1(d)+\ldots+\delta_m(d)=\delta(d)} \left\{ \max_{i=1..m} \{\varepsilon_i\} \right\} . \tag{5.16}$$

While it is difficult to determine the best choice of the partitioning values $\delta_i(d)$, a good heuristic choice (that is guaranteed to yield correct bounds) is to set

$$\delta_i(d) \quad = \quad \delta(d) \cdot \frac{high_i}{\sum_{i=1}^{m} high_i} \tag{5.17}$$

(i.e., we choose the $\delta_i(d)$ values in proportion to the $high_i$ values of the index lists at the current scan positions). Computing these generalized Chernoff-Hoeffding bounds can be programmed as OpenMaple procedures as well (see Appendix A.2 for an OpenMaple implementation of these estimators).

As the computation of convolutions and their bounds using OpenMaple involves the derivation of various density functions and finding roots numerically (e.g., using the Monte-Carlo [PFTV92] method), our implementation incurs a non-negligible overhead, part of which is caused by the communication overhead and iterative invocation of different software environments such as Java, the Java Native Interface (JNI) for C libraries, and OpenMaple itself. Despite the flexibility in applying Chernoff-Hoeffding bounds to different kinds of distributions and even mixtures of heterogeneous distributions, we are also interested in approximations that are computationally cheaper.

5.2.2 Poisson Estimators

Another form of distribution that has nice theoretical properties, can be efficiently evaluated, and is a reasonable fit for realistic score distributions (e.g., the BM25-based score distributions for terms in large corpora) is the *Poisson distribution*. In order to fit the real score distribution of an inverted list L_i, we assume that S_i is a discrete random variable with n equidistant values $\nu_j = 1 - j \cdot \frac{high_i}{n}$, for $j = 0, \ldots, n-1$, i.e., we discretize all scores $s_i(d)$ for $d \in L_i$ according to a tunable parameter n (e.g., between 20 and 100) that denotes the resolution of the discretization and affects the accuracy of the predictor. Then the probability for an object having a local score ν_k is

$$P\left[S_i = \nu_k\right] \quad = \quad e^{-\lambda_i} \frac{\lambda_i^k}{k!}. \tag{5.18}$$

Here, λ_i is the parameter that we fit to the actual distribution. The particularity of the Poisson distribution is that the convolution of m such distributions with parameters $\lambda_1, \ldots, \lambda_m$ is again a Poisson distribution with parameter $\lambda = \lambda_1 + \ldots \lambda_m$. As the $high_i$ values change during the index scans, we actually need to predict $P\left[\sum_{i \in \bar{E}(d)} S_i > \nu \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right]$, where ν is the largest value smaller than the relevant $\delta(d)$ in the virtual value discretization.

$$p(d) \quad = \quad P\left[\sum_{i \in \bar{E}(d)} S_i > \nu \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right] \tag{5.19}$$

$$= \quad 1 - P\left[\sum_{i \in \bar{E}(d)} S_i \leq \nu \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right] \tag{5.20}$$

$$= \quad 1 - \frac{P\left[\sum_{i \in \bar{E}(d)} S_i \leq \nu \ \bigwedge \ S_i \leq high_i \ for \ i \in \bar{E}(d)\right]}{P\left[S_i \leq high_i \ for \ i \in \bar{E}(d)\right]} \tag{5.21}$$

$$\tag{5.22}$$

Again, assuming independence among the random variables S_i, we lower-bound this probability as follows:

$$\geq \quad 1 - \frac{P\left[\sum_{i \in \bar{E}(d)} S_i \leq \nu\right]}{\prod_{i \in \bar{E}(d)} P\left[S_i \leq high_i\right]} \tag{5.23}$$

$$= \quad 1 - \frac{\sum_{k=1}^{\nu \cdot n} \left(\frac{1}{k!} \ e^{-\sum_{i \in \bar{E}(d)} \lambda_i} \ \left(\sum_{i \in \bar{E}(d)} \lambda_i\right)^k \right)}{\prod_{i \in \bar{E}(d)} P\left[S_i \leq high_i\right]} \tag{5.24}$$

$$\tag{5.25}$$

Here, $\nu \cdot n$ denotes the number of the bucket for the discretized ν value in the convoluted Poisson predictor over $m' = |\bar{E}(d)|$ remainder dimensions. Instead of summing up the $\nu \cdot n$ first terms of the Poisson probability function to get the cumulated probability

$$P\left[\sum_{i \in \bar{E}(d)} S_i \leq \nu\right] \quad = \quad \sum_{k=1}^{\nu \cdot n} P\left[\sum_{i \in \bar{E}(d)} S_i = \nu_k\right] \ , \tag{5.26}$$

we use the efficient numerical method given in [PFTV92] based on the *Incomplete Gamma Function*, which converges rapidly for $\nu \cdot n$ less than about $k + 1$ and provides a very good approximation of the cumulative Poisson probability function; and the individual $P\left[S_i \leq high_i\right]$ can easily be derived from the basic Poisson estimators for the individual lists without having to consider the convolution.

5.2.3 Histograms

Real score distributions may sometimes be impossible to capture with basic distribution functions and parameter fitting. In such cases, the only viable solution is to explicitly track the distribution in the form of a compact *histogram* [Ioa03]. Since histogram construction is not exactly inexpensive, we precompute a histogram for the score distribution of each index list offline, either by fully scanning entire inverted lists or by sampling the score entries. At query time, we first compute the convolution of the query-relevant histograms (and possibly of subsets of them). For simplicity, we consider only equi-width histograms, but our approach could be easily generalized

to more sophisticated histogram variants (see [Ioa03] for an overview), at higher precomputation and runtime costs, however. In order to break predictor ties among the entries of each histogram bucket, we assume a Uniform distribution of scores within each bucket. For conservative probabilistic predictions, we might alternatively assume that all values within one histogram cell coincide with the upper bound of the cell.

Similarly to the Poisson estimator described beforehand, we discretize the input score domains and choose the same number n of cells for each basic histogram, thus covering the score range $(0, 1]$ and we use $m' \cdot n$ cells for the convolution histogram over m' basic histograms, with the same width $1/n$ as the basic histograms, thus covering the range $(0, m']$. This way, cell i (for $i = 0, \ldots, n-1$ or $i = 0, \ldots, m \cdot n - 1$) covers the interval $(lb_i, ub_i]$ with $lb_i = i/n$ and $ub_i = (i+1)/n$. Each cell stores the frequency $freq_i$ and the cumulative frequency $cfreq_i$ of scores that fall into its interval. Then the convolution H of basic histograms $H_1, \ldots, H'_m$ is computed by

$$H.freq_i = \sum_{\substack{(i_1, \ldots, i_m) \\ with \\ \sum_l^{m'} i_l = i}} H_1.freq_{i_1} \cdot \ldots \cdot H'_m.freq_{i'_m} \tag{5.27}$$

and

$$H.cfreq_i = \sum_{j=0}^{i} H.freq_j \ . \tag{5.28}$$

The above formula suggests some form of dynamic programming for convolutions over arbitrary remainder dimensions using m' nested loops. Since this would render the convolution procedure infeasible for high-dimensional queries, we split the convolution computation in a series of binary convolutions, where each of the m' histograms is iteratively combined with the previous convolution in an overall time of $O(m'n^2)$. The computation of the probability then simply resolves in looking up the cumulated frequency in the convolution histogram for the respective value of $\delta(d)$:

$$P\left[\sum_{i \in \bar{E}(d)} S_i > \delta\right] = (1 - H.cfreq_i) \ with \ \delta \in (lb_i, ub_i] \tag{5.29}$$

Algorithm 6 shows pseudo code for a binary convolution procedure with two nested loops that merge the relative bucket frequencies for two input histograms. Exploiting the commutativity of convolutions, we can reduce the m-dimensional case to a series of these binary convolutions in arbitrary order.

The full convolution histogram that we compute for a given query captures the complete distribution ($m' = m$) of possible global scores and partial scores over unevaluated dimensions (here, we compute the convolution of the basic histograms for all $m' = |\bar{E}(d)| = m - |E(d)|$ unevaluated dimensions). Note that we can of course reuse convolution histograms for an arbitrary amount of candidate tests in the index scans, but we may have to compute up to $2^m - 1$ convolutions corresponding to the possible amount of distinct remainder sets that candidates can be grouped into.

Algorithm 6 Binary convolution procedure.

```
getBinaryConvolution(Histogram a, Histogram b)
convolution.freq[0..a.n+b.n]   := 0;
convolution.cFreq[0..a.n+b.n] := 0;
for i := 0; i < a.n; i++ do
   for j := 0; j < b.n; j++ do
      convolution.freq[i + j] += a.freq[i] · b.freq[j];
   end for
end for
c := 0;
for i := 0; i < convolution.n; i++ do
   c += convolution.freq[i];
   convolution.cfreq := c;
end for
return convolution;
```

Furthermore, we would like to periodically update these convolutions. As the index scans proceed, we are actually interested in conditional probabilities of the form $P\left[\sum_{i \in \bar{E}(d)} S_i > \delta \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right]$, where the $high_i$ values reflect the current positions in the index scans. Obviously, dynamically rebuilding the histograms after every sorted access is out of the question. We have three ways of addressing this point. The first option is to conservatively bound the conditional probability, analogously to the Poisson approximation model:

$$p(d) = P\left[\sum_{i \in \bar{E}(d)} S_i > \delta \mid S_i \leq high_i \; for \; i \in \bar{E}(d)\right] \tag{5.30}$$

$$\geq 1 - \frac{P\left[\sum_{i \in \bar{E}(d)} S_i \leq \delta\right]}{\prod_{i \in \bar{E}(d)} P\left[S_i \leq high_i\right]} \tag{5.31}$$

which can be directly looked up in the precomputed histograms.

The second option is to start with the full convolution histograms and dynamically "undo" the terms that contribute to $H.freq_i$ as the $high_j$ values

change during query execution. Suppose that $high_j$ changes from some value ub_k to ub_{k-1}. Then we modify all $H.freq_i$ values with $\nu_i \leq \sum_{j=0}^{m'} high_j$ as follows:

$$H.freq_i \;=\; H.freq[i] \;-\; \sum_{\substack{(i_1,\ldots,i_m)\\ with\\ \sum_l^{m'} i_l = i\\ and\\ i_l = k}} H_1.freq_{i_1} \cdot \ldots \cdot H'_m.freq_{i'_m} \tag{5.32}$$

The subtrahend is also precomputed and additionally stored in cell k of the histogram for index list L_l. The computational overhead for the dynamic maintenance is $O(m'\,n)$ whenever one of the index scans crosses a histogram cell boundary, but the – less critical – precomputation cost and the space for each histogram increase considerably (with $O(m'^2\,n^2)$ space instead of $O(m'\,n)$).

Finally, the third way is to periodically recompute the histograms, after every b sorted accesses with b being in the order of a few thousand. Each time a convolution histogram is rebuilt from the precomputed basic histograms, the current $high_i$ values are taken into account; so the recomputation becomes cheaper and the range of valid histogram buckets becomes smaller as the index scans proceed toward lower local scores. We found this to be the best choice for the trade-off between runtime and space required.

5.2.4 Extensions and Generalizations

Our framework for probabilistic predictions could be extended in the following ways:

1) supporting more general score aggregation functions other than summation,

2) adding further classes of score distributions for specific scoring models,

3) adding selectivity estimators that also take the length of the inverted lists into account, and

4) investigating the influence of correlated local scores.

Various Score Aggregations

Using monotonous score aggregation functions beyond simple sums is already supported, to a large extent, within our framework. As described in Section 3.1.2, a large class of aggregation options can simply be cast into the precomputation of local scores, so that the actual aggregation step again becomes a simple summation. For example, with weighted summation the weights for each dimension can be factored into the local scores; IR-style TF·IDF-based scores are of this type, since IDF values can be viewed as dimension weights. Also note that *Cosine* similarity in IR is usually reduced to summation (i.e., scalar products between document and query vectors) by normalizing all document vectors to length 1 using L2 norm. Similarly, the *product* can be cast into a simple summation over log-transformed document and query weights. Using the *maximum* for score aggregation is even simpler than summation; instead of computing the convolution of several S_i distributions, we merely compute $P[\max_i\{S_i\} > \delta] = 1 - \prod_i P[S_i \leq \delta]$. Section 7.6 discusses the materialization of meta histograms that capture this max-distribution for multiple merged lists in the context of query expansion.

Various Score Distributions

As for score distributions, we can accommodate a wide variety of distributions into the Chernoff-Hoeffding bound approach discussed in Subsection 5.2.1. For example, it would be straightforward to incorporate Zipf distributed scores, where $P[S_i = \nu_k]$ for equidistant values ν_k is proportional to $1/k$ and the cumulative distribution corresponds to the harmonic series, and we can also easily handle heterogeneous mixes of different distributions, say Uniform for some index lists but Zipf for some highly skewed ones. Moreover, our experiments indicate an effective pruning behavior of the Poisson estimator for more skewed distributions such as real-world TF·IDF-style scores, or even artificially generated Zipf distributions, with a good trade-off between retrieval quality and efficiency. 2-Poisson mixes would be another intriguing option, taking the probabilistic arguments for the BM25 scoring model into account, with similar closed-form solutions for the convolution. For the histogram approach of Section 5.2.3, more general distributions are a non-issue, because histograms are approximations of arbitrary distributions.

Selectivity Estimations

The score predictor implicitly assumes that a document occurs in all its missing dimensions, hence it inherently overestimates the probability that a document can get a score higher than the current *min-k*. For a more precise

estimation, we would like to also take the selectivity of the lists into account, i.e., the probability that a document occurs in the remaining part of a list. A simple extension to our histogram approach would be to just add an additional 0-scored bucket with the frequency $\frac{n-|L_i|}{n}$ of all documents that are not contained in L_i. Unfortunately, this 0-scored bucket would dominate all the other buckets in the basic input histograms as well as in the convolution histograms, yet it would be difficult to incorporate this approach also into our closed-form score predictors, using equi-width histograms, Chernoff-Hoeffding bounds, or through fitting a parameterized Poisson predictor. We rather postpone the solution at this point and will come back to this issue in Section 6.2.2, where we discuss a combined score predictor and selectivity estimator.

Feature Correlations

As for correlations between the local scores from different index lists, the generalized Chernoff-Hoeffding bounds already provide an approach. The histogram approach, on the other hand, would have to use multidimensional histograms to capture joint distributions. We are not convinced that this is practically viable except for specialized settings. Multidimensional histograms over all index lists may be very space-consuming and either sparse or inaccurate, and the subspace that is relevant for a given query is known only at query time when histogram building would already be part of the user-perceived response time. Fitting a parameterized multidimensional distribution, e.g., a multivariate Normal distribution, to the data seems more promising, but the decision for a particular type of distribution function would have to be carefully justified. Section 6.2.3 also provides a more general solution to this issue, where feature correlations are captured by a more sophisticated selectivity estimator that may in turn be combined with our score predictor approach. Our experiments indicate that for probabilistic candidate pruning, the score predictor approach already discriminates different candidate items reliably and provides a versatile building block for highly efficient top-k query processing with a controllable loss in result precision and recall.

5.3 Efficient Queue Management

Our query processing algorithms use the probabilistic models as predictors for the global scores of data objects that have not been fully evaluated or not seen at all in the index scans so far. Based on this central building block, we

have developed several algorithms that differ in their selection of candidates to which they apply the probabilistic predictions, as well as in their actions that they take when a threshold test for a candidate fails, i.e., the candidate is unlikely to be able to qualify for the top-k result. All algorithms maintain the set of current top-k objects and the set of candidates organized as a hash table based on object ids.

5.3.1 Conservative Algorithm

A naive algorithm would simply predict the scores of all candidate objects in every step of the index scans and drop all candidates whose probabilities of qualifying for the top-k result are sufficiently low. This would incur very high overhead for probabilistic threshold tests; moreover, the score prediction for an object d would have to be recomputed whenever one of the $high_i$ values in the set $\bar{E}(d) = \{1..m\} - E(d)$ changes. A better way is to group the candidates by their $E(d)$ sets, placing all objects with the same set of evaluated dimensions into one partition using $bestscore(d)$ as priorities. We will refer to this pruning strategy as the Conservative Algorithm (or `Prob-cons` for short).

Then it suffices to test only the best object per group, i.e., the one with the highest predicted score. This object dominates all other candidates in the same group in terms of the probability of qualifying for the top-k result. Across groups, however, the top objects are not directly comparable. This strategy benefits from the observation that in the beginning of the index scans the distribution of remainder sets is scattered across different lists and groups of remainder dimensions, but very quickly after the pruning of candidates dominates the amount of newly discovered candidates, namely at the latest when $\sum_i high_i \leq min\text{-}k$. Then there are only very few dominating groups consisting of the longer index lists left, each with the potential to prune a major amount of remaining candidates with only a single probabilistic threshold test.

Based on this consideration, the conservative algorithm maintains a priority queue for each subset $E(d) \subseteq \{1..m\}$, i.e., up to $2^m - 1$ queues for a query with m specified conditions. Each of the queues merely contains pointers to the hash table entries of the candidate objects. Note that m is much smaller than the dimensionality of the data space (e.g., for keyword queries over a text document space).

As an item d is evaluated at the current scan position pos_i in index list L_i, the conservative algorithm deletes d from the $\bar{E}(d)$ queue and, if $worstscore(d)$ still fails the threshold $min\text{-}k$, inserts it into the $E(d) \cup \{i\}$ queue using its updated $bestscore(d)$ as priority; the insertion is possible

with cost $O(\log n)$ using a binomial heap, or with amortized cost $O(1)$ using a Fibonacci heap [CLRC01]. If $E(d) \cup \{i\} = \{1..m\}$, i.e., d is completely evaluated, then d is dropped from the candidate list. In this case, d's score bounds have converged to $worstscore(d) = bestscore(d)$; and either d exceeds the *min-k* threshold such that d is added to the top-k list, or it is not a relevant candidate any more. If an index list L_i has been completely scanned down to its end, we set $high_i = 0$ and may safely remove i from all items' remainder dimensions and update their bestscores, which in turn triggers the pruning of many candidates.

For periodic index pruning, e.g., after every $b = 1,000$ index scanning steps, the top elements of all queues are probabilistically tested against the current threshold *min-k*. When a top element fails the test, then all elements of that queue are dropped from the candidate list. The algorithm proceeds with fewer candidates and eventually stops when all queues have become empty. Note that there is also one queue for $E(d) = \emptyset$ with a single *virtual candidate*, thus capturing the predicted score for an object that has not been seen at all so far.

Advancing the scan pointer in one index list may affect the priorities of other candidates, too, namely by possibly reducing the $high_i$ value of one dimension. But within each queue, this change will affect all candidates in the same way; so we can simply track $\sum_{i \in \bar{E}(d)} high_i$ per queue in constant time $O(1)$ as basis for pruning.

5.3.2 Aggressive Algorithm

The Aggressive Algorithm (`Prob-aggr`) is the extreme opposite of the Conservative Algorithm. It considers one candidate object for probabilistic testing, only, namely the virtual document d with $E(d) = \emptyset$, $worstscore(d) = 0$, and $bestscore(d) = \sum_{i=1}^{m} high_i$ consisting of the upper bounds $high_i$ at the current scan positions. If the score prediction for this object falls below the threshold *min-k*, the algorithm stops immediately. Since the prediction for this unknown object is exclusively based on the $high_i$ scores at the current scan positions, this item's bestscore yields an upper bound for all yet unseen documents. That is, even without probabilistic pruning this algorithm typically stops before the truly best candidate would fail the *min-k* threshold and, thus, yields an approximate result even without the probabilistically relaxed threshold termination. Similarly to a real candidate document, we may also reason about the probability $P[\sum_i S_i > \textit{min-k} \mid S_i \leq high_i]$ of the virtual candidate to get into the top-k results as an additional accelerating factor for query processing and algorithm termination, even before $\sum_i high_i$ really falls below *min-k*. The strength of the aggressive algorithm is its min-

imal overhead, but we do not expect it to perform too well in terms of result precision. Moreover, because of its more aggressive pruning behavior with only a single candidate to be tested probabilistically, this method is less robust and will scale worse with the probabilistic pruning parameter ε, with major potential drops in result precision and runtime for individual queries.

5.3.3 Progressive Algorithm

In between the overly eager behavior of the `Prob-aggr` and the substantial queuing overhead of `Prob-cons` is the Progressive Algorithm (`Prob-prog`) which maintains a single priority queue for all candidate objects. Again, the queue elements are prioritized in descending order of bestscores. In each step, the priority of the current candidate, i.e., the one previously fetched from the index list, is updated and the queue is maintained accordingly. Note that the major amount of candidates is typically not rediscovered and updated, though, and would get "stuck" in the queue for an overly long duration.

The algorithm is conservative, because it does not immediately track the bestscore changes that result from reduced $high_i$ values in each step, but leaves the bestscore values higher than they actually are. Otherwise all queue elements would have to be updated after each sorted access step which would in fact lead to rebuilding the entire queue. An additional implementation trick that we employ is that the queue is periodically traversed, e.g., again after every $b = 1,000$ index-scanning steps, and we tentatively compute the up-to-date bestscore values of each queue element based on the current $high_i$ values. All elements that do no longer pass the threshold test are dropped from the queue. The priorities of the "surviving" elements are not updated to avoid a massive batch of queue operations. So this periodic removal of unneeded queue elements can be seen as a kind of *garbage collection* without having to rebuild the entire queue.

In conjunction with the periodic garbage collection, the progressive algorithm invokes the probabilistic predictor for each element d of the queue using its up-to-date $bestscore(d)$. All objects that fail this probabilistic threshold test are dropped from the queue. The algorithm stops when its queue becomes empty or the top element's bestscore falls below the threshold $min\text{-}k$.

Smart Algorithm

`Prob-prog` could stop earlier, if it reconsidered all elements in the priority queue with the changing $high_i$ values reflected in each step. In the Smart Algorithm (`Prob-smart`), we periodically rebuild the entire priority queue of current candidates with the currently known $high_i$ values taken into consider-

ation. By default, the queue is rebuilt every $b = 1,000$ steps. The rebuilding has amortized cost $O(n \log n)$ for n queue elements, using a Fibonacci or a Binomial heap [CLRC01]. For an online algorithm operating on very large index lists this cost may still be out of the question. Therefore, the smart algorithm maintains only a bounded priority queue. Whenever it is rebuilt, only the best q elements are kept, with q being in the order of a few hundred or thousand. Newly encountered data objects are admitted to enlarge the queue until the next rebuild; so the maximum size is actually $q+b$, but every rebuild truncates the size back to q.

As the priority queue is fully up-to-date after every rebuild, the smart algorithm can take more aggressive actions than the progressive method with regard to candidate pruning. If the top element of the rebuilt queue does not pass the probabilistic threshold test, the smart algorithm immediately stops all index scans and terminates.

5.3.4 Common Framework

All four algorithms share the same algorithmic skeleton illustrated for the TopX core query processor in Algorithm 3 (see Section 4.3). We refer to the above queuing extensions as the `Prob-`k family of algorithms. Note that, in addition to the probabilistic predictions and corresponding probabilistic threshold tests, all algorithms also include the original *min-k* threshold test to compare the maximum bestscore among all candidates against the worstscore of the current top-k objects. If this test fails, all index scans can be stopped immediately, and this extra test is so lightweight that we can always include it after each sorted access.

Candidate Queuing vs. Candidate Pooling

The usage of a separate (and optionally bounded) priority queue for maintaining the candidate pool yields an up-to-date view on the (top) candidates that are not currently among the top-k results and allows for a cheap *min-k* threshold test after each single index access and candidate update just by testing the top-priority item from the queue in constant time $O(1)$. In particular, this allows us to immediately terminate query processing in-between two batches of sorted access, however, at the expense of having to continuously maintain a sorted candidate list in memory with amortized update cost of $O(\log q)$ per index access and candidate update. If we set aside this aspect and allow for a slightly coarser stopping condition, we may just omit continuous queue updates and only test the threshold condition in conjunction with each of the iterative thread synchronizations and queue rebuilds.

Since all candidates' priorities can only be updated iteratively for a feasible implementation and therefore tend to be slightly overestimated anyway, we might be willing to trade a small increase in access rates of at most $b - 1$ steps for saving a significant amount of computation cost.

Hence, as an alternative to the conservative queuing approach, we may merely maintain an unsorted list as *candidate pool*, e.g, by iteratively polling the whole cache, with iterative candidate updates and linear scans for a single, currently top-ranked candidate, with linear cost in the number of candidates $O(q)$. However, for the experiments with a primary focus on saved access rates, the default queuing strategy of the Prob-sorted family of algorithms was used.

As for the probabilistic pruning, we merely mark candidates that fail the probabilistic threshold test and only drop them from the queue but *not* from the cache. This significantly increases the robustness of the probabilistic pruning, as it ensures that candidates that might have been falsely pruned due to predictor mistakes are still "remembered" and have a chance to get into the top-k list at a later point of the index scans, while early algorithm termination due to a reduced queue size is assured.

Choosing the Queue Bound

Using a bounded queue as proposed by the Smart algorithm implies turning the safe stopping criterion as specified by the original TA algorithm into a heuristic also when no probabilistic candidate pruning is used, since an eagerly small choice of the queue bound q might make the algorithm run out of candidates too early and return approximate top-k results. Using an unbounded queue on the other hand would render the procedure extremely inefficient with most of the actual query processing time being spent on updating and sorting candidates.

In order to determine a safe choice for the queue bound, let us therefore consider the following construction: The task of finding the top-k results can be reformulated as to finding the score of the final $(k + 1)^{th}$ candidate $\widetilde{d}$, i.e., the best candidate that just does not make it correct top-k result any more. As we keep on scanning the inverted lists, we have to remember all candidates that are still eligible for the top-k in some kind of memory or queue. Now, $\widetilde{d}$ has to be detected at an early point of the sequential scans, because it must have a bestscore that makes it a top candidate until right before algorithm termination, since its bestscore yields the counterpart for the *min-k* threshold for stopping. The thing is that we do not know $\widetilde{d}$ in advance, and moreover, it may even temporarily move through the top-k list and be pushed back into the candidate queue by some other top-k items.

Such a top-k pass-through of $\widetilde{d}$ can happen at most b times during a batch of b index accesses. Then, the queue is truncated back to the queue bound q. Assuming no ties in local scores, $\widetilde{d}$ is either in the intermediate top-k or among the top-b candidates at any time of the query processing, such that $q \geq b$ is a safe choice for truncating the queue. With ties, however, this bound might degrade. Experiments for the TREC GOV collection (see Section 9.7.1) with queue bounds q down to a size of 100 (and a batch size $b = 200$) did not exhibit any approximative results due to the bounded queue heuristic – but it did substantially decrease the amount of queuing overhead and, thus, contribute in much better query runtimes.

NRA Baseline Implementation

Finally, we comment on our implementation of the NRA baseline algorithm. The original papers on TA and NRA do not specify any concrete data structures for the candidate set and how to determine the best candidate in each step. We decided to implement these aspects analogously to the `Prob-prog` algorithm, by maintaining a single priority queue with bestscore values as priorities. As for implementing the priority queue, we chose a Fibonacci heap with very efficient amortized insertion cost of $O(1)$ and logarithmic update cost $O(\log q)$. Like before, we do not update all queue elements whenever one of the $high_i$ values changes, but only update the element currently encountered in the index scan and merely perform periodic garbage collections on the queue with tentative updates.

5.4 Top-k Guarantees for Probabilistic Candidate Pruning

The previous considerations provide us with score predictions for individual candidate items at arbitrary steps during the sequential index accesses. These probabilistic predictions in our query processing strategies lead to probabilistic guarantees from a user viewpoint, if we restrict the action upon a failed threshold test to dropping candidates but stop the entire algorithm only if we run out of candidates. This is the situation given in the `Prob-cons` and `Prob-prog` algorithms. In this case the probability of missing an object that should be in the true top-k result is the same as erroneously dropping a candidate, i.e., pruning errors are assumed to be uniformly distributed among all items discovered during index processing; and this error, call it p_{miss}, is bounded by the probability ε that we use in the probabilistic predictor when assessing a candidate. For the *relative* recall of the top-k result, i.e., the frac-

tion of true top-k objects that the approximate method returns, this means that

$$
\begin{aligned}
P[recall = r/k] &= && (5.33)\\
P[precision = r/k] &= && (5.34)\\
&= \binom{k}{r}(1-p_{miss})^r\, p_{miss}{}^{(k-r)} && (5.35)\\
&\leq \binom{k}{r}(1-\varepsilon)^r \varepsilon^{(k-r)} && (5.36)
\end{aligned}
$$

where r denotes the number of correct results in the approximate top-k. We can then efficiently compute Chernoff-Hoeffding bounds for this Binomial distribution. Note that the very same probabilistic guarantee holds for the precision of the returned top-k result, simply because recall and precision use the same denominator k in this case. The predicted expected precision then is

$$
\begin{aligned}
E\,[precision] &= \sum_{r=0}^{k} P\,[precision = r/k] \;\cdot\; \frac{r}{k} && (5.37)\\
&= 1-\varepsilon\ . && (5.38)
\end{aligned}
$$

For the more heuristic strategies, `Prob-smart` and `Pro-aggr`, that only test top elements of priority queues and, upon a failed probabilistic threshold test, stop the entire algorithm, carrying over the candidate-error probability ε to an argument about recall and precision guarantees would be more sophisticated. Our experiments for the `Prob-cons` and `Prob-prog` query evaluation strategies using histogram convolutions over a basic TF·IDF model in fact reveal the relative precision/recall curves almost as linearly decreasing functions of ε which confirms the bounds suggested above.

This result yields a compact and intuitive assumption on the result quality that the approximate top-k algorithms provides compared to the exact top-k algorithm without probabilistic pruning in terms of *relative* precision or recall, i.e., the overlap of two result sets. In practice, the score differences between the top-ranked items are often very marginal for many real-world datasets and scoring models such as TF·IDF or BM25. Moreover, all the scoring models developed in IR which are aiming to model user-perceived relevance as a compact score aggregation, often using term independence assumptions, have a heuristic nature themselves. Our experiments on various data collections using human relevance judgments for query results indicate that with increasing pruning aggressiveness, the user-perceived result quality decreases at a much lower rate than the relative overlap measures, especially

in typical large-corpus benchmark settings such as TREC or INEX, where absolute recall for a top-k query, with k being in the order of 10 or 20, is not a critical issue. Often the top-k results exhibit significant changes through probabilistic pruning and early algorithm termination, but the user-perceived precision measures remain almost equally high.

Chapter 6

Index Access Scheduling

Approximation may be a viable choice from an IR point-of-view, with almost arbitrary runtime gains, however, at the price of a noticeable loss in result precision and recall. In this chapter, we investigate on how to exploit very similar statistics, compared to those employed for probabilistic candidate pruning, for accelerating query executions with no loss in result quality, thus following our work developed in [BMT+06]. The continuous optimization of top-k queries entails scheduling for the two kinds of accesses:

1) the prioritization of different index lists in the sorted accesses (SA), and

2) the decision on when to perform random accesses (RA) and for which candidates.

The chapter develops an integrated view of SA and RA scheduling issues and presents novel strategies that outperform prior state-of-the-art proposals such as Fagin's Combined Algorithm (CA) [FLN01] algorithm by a large margin. Our main contribution are new, principled scheduling methods based on a Knapsack-related optimization for sequential accesses and a beneficial cost model for random accesses. The methods can be further boosted by harnessing probabilistic estimators for scores, selectivities, and index list correlations.

Albeit the performance gains of these algorithms is significantly lower than for the probabilistic pruning approach, our experiments show that our proposed methods achieve significant gains compared to the CA baseline on three different datasets (the TREC Terabyte text collection, a semistructured version of the IMDB movie collection, and a large, highly structured collection of HTTP server logs for the 1998 soccer worldcup): a factor of up to 3 in terms of abstract execution costs, and a factor of 5 in terms of absolute

runtimes of our implementation. We also show that our best techniques are within 20 percent of an empirically evaluated lower bound for the execution cost of any top-k algorithm from the TA family; so our probabilistic cost model gets fairly close to an optimal scheduling.

6.1 Index-Optimized Top-k Query Processing

6.1.1 Adaptive Index Access Scheduling

The potential cost savings for flexible and intelligent scheduling of index-scan steps result from the fact that the descending scores in different lists exhibit different degrees of skew and may also be correlated across different lists. For example, dynamically identifying one or a few lists where the scores drop sharply after the current scan position may enable a TA-style algorithm to eliminate many top-k candidates much more quickly and terminate the query execution much earlier than with standard round-robin scheduling or the best compile-time-generated plan. These savings are highly significant when index lists are long, with millions of entries that span multiple disk tracks, and the total data volume rules out a solution where all index lists are completely kept in memory (i.e., with multi-Terabyte datasets like big data warehouses, Web-scale indexes, or internet archives).

Adaptive Query Executions

As the local characteristics (i.e., the *skew*) of score distributions may vary across index lists and with increasing scan depths, the SA scheduling decisions have to be *adaptive*, thus taking the currently best combination of individual sorted access batches into account, in order to lower the bestscores of many candidates to the highest possible extent and with the lowest effort in terms of additional scan depths required. This way, the SA scheduling has to be aware of both the distribution of scores in the upcoming parts of the index lists (which calls for the usage of histograms) and of the benefit of these scheduling decisions for each individual candidate in the queue, in order to eliminate as many candidates as possible and, thus, stop earlier.

RA scheduling, on the other hand, has to take each individual candidate object into account to determine if and when it is cost-beneficial to explicitly schedule a random access for that particular candidate in order to determine its final score. It therefore compares the cost of looking up the candidate, thus paying the price for one or more expensive random accesses to disk-resident index structures, with the (expected) cost of not looking up that candidate, and thus having to keep the candidate in the queue, waiting for

it to be pruned or to get promoted to the intermediate top-k results during another batch of sorted accesses. In any case, scheduling a random access definitely affects the dynamics of the system, since the candidate will be removed from the queue after the random lookup. Either it is pruned and ultimately dropped when its final score falls below the *min-k* threshold, or it is promoted to the top-k list. In the first case, the algorithm might stop earlier, because the queue gets smaller; in the second case, taking another item into the top-k results means increasing the *min-k* threshold which may in turn lead to an increased pruning of candidates from the queue.

Hence, the ultimate goal is to accelerate query executions and, at the same time, limit or even aim to reduce the memory consumption for candidate queues and other auxiliary data structures. The statistics that we consider in this context are histograms over the score distributions of individual index lists and also the (precomputed) correlations between index lists that are processed within the same query.

Adaptive Environment Customization

Moreover, the presented scheduling algorithms are also adaptive to different system environments and middleware layers. Given an arbitrary cost ratio c_R/c_S for the cost of a single random versus a single sorted access, which can easily be empirically measured or even be set up to be self-tuning, we can easily adapt the generated scheduling decisions, e.g., using a cost ratio of 50–100 for a top-k engine on top of a DBMS, with relatively low sequential throughput but good random access performance through caching, or 500-1,000 for inverted files and direct disk access providing faster sequential access and slower random access performance, or lastly 1,000–10,000 for a distributed system with high network latency. Note that $c_R/c_S = \infty$ would generate a plan that just corresponds to the NRA baseline algorithm; and $c_R/c_S = 0$ corresponds to the original TA (see [Fag99] for both).

6.1.2 Extended Classification of Threshold Algorithms

Based on the previous considerations, let us first extend our initial taxonomy of top-k selection queries as introduced in Section 1.2. Different algorithmic instances within the paradigm of top-k selection queries with support for both sorted and random access differ in the ways how they handle three fundamental issues:

1) how sorted accesses are scheduled,

2) how random accesses are scheduled, and

3) how random accesses are ordered.

This subsection presents an extended taxonomy of the different possibilities for each dimension and classifies the existing approaches (TA, NRA, CA) in this scheme and points out the new approaches presented here.

Sorted Access Scheduling

- `RR`: Round-robin (TA, see [FLN03, GBK01, NR99], NRA, see [FLN03, GBK01], and CA, see [Fag02]), see also Section 1.4.1.
- `KSR`: Knapsack-based optimization of `RR` that aims to maximize the reduction of scores at the future scan positions for a fixed batch of sorted accesses (see Section 6.3.1).
- `KBA`: Knapsack-based optimization of `RR` that aims to maximize an aggregated benefit among all candidates currently being in the queue for a fixed batch of sorted accesses (see Section 6.3.2).

Random Access Scheduling

- `Never`: Perform SA only (NRA, see [FLN03, GBK01]).
- `All`: After each SA, perform full RA for each new candidate to retrieve its final score; no candidate queuing is required (TA, see [FLN03, GBK01, NR99]).
- `Each`: After each round of SAs, schedule a balanced amount of RAs according to the current cost ratio c_R/c_S between RAs and SAs performed so far (CA, see [Fag02]).
- `Last`: We start with performing only batches of SAs and, at some point in the algorithm, we switch to performing only RA, thus scheduling the full amount of RA to eliminate all the remaining items in the queue (see Section 6.4.1 and 6.4.2). We stop the SA batches according to the *estimated* cost for the remaining RAs (i.e., corresponding to the estimated number of candidates in the queue that need to be looked up to raise *min-k* above the bestscore of the currently best candidate).

Random Access Ordering

- `Best`: Perform RAs in descending order of $bestscore(d_j)$ (CA, see [Fag02] and Section 6.4.1).

- `Ben`: Perform RAs according to a cost model, i.e., proportionally to the probability $p(d_j)$ that d_j gets into the top-k results (see Section 6.4.2).

Any algorithm for TA-style top-k query processing now corresponds to a triplet, for example, the NRA scheme from [FLN03] corresponds to `RR-Never`, TA corresponds to `RR-All`, and CA is `RR-Each-Best`. Obviously, only certain RA-scheduling and -ordering combinations make sense, whereas any SA-scheduling approach may be combined with a given RA-scheduling and -ordering combination. In Sections 6.3 and 6.4 we will investigate the more sophisticated combinations. Our best results have been obtained by the combination `KSR-Last-Ben`.

6.2 Probabilistic Extensions

In this section, we refine the details for estimating the probability $p(d)$ that a candidate document d with non-empty remainder set $\bar{E}(d)$ may qualify for the top-k results. Recall from the previous chapter that the way how we estimate $p(d)$ depends on the assumptions that we make about the distribution of unknown scores that d would obtain from each remaining list in $\bar{E}(d)$. For each missing dimension, we consider a random variable S_i for the score of d in that dimension. Then we estimate the probability $p_S(d)$ that a candidate document can get enough score mass from its remaining lists to enter the top-k as

$$p_S(d) := P\left[\sum_{i \in \bar{E}(d)} S_i > \delta(d) \mid S_i \leq high_i\right] \tag{6.1}$$

As this involves the sum of random variables, this entails computing the convolution of the corresponding distributions to compute this probability, using either a parameterized score estimator or compact and flexible histograms. Then we can utilize $p_S(d)$ as an approximation of $p(d)$ (as deduced from Section 5.2).

6.2.1 Selectivity Estimator

The score predictor implicitly assumes that a document occurs in all its missing dimensions, hence it inherently overestimates the probability that a document can get a score higher than the current *min-k* threshold. For a more precise estimation of the probability, we take the selectivity of the lists into account, i.e., the probability that a document occurs in the remaining

part of a list. For a single list L_i with length l_i and a total dataset size of n documents, this probability is

$$q_i(d) := \frac{l_i - pos_i}{n - pos_i} \ , \tag{6.2}$$

where pos_i denotes the current scan position in list L_i. For a partially evaluated document d with a set $\bar{E}(d)$ of remainder dimensions, the probability $q(d)$ that d occurs in at least one of the dimensions in $\bar{E}(d)$ is computed as

$$\begin{aligned} q(d) &:= P\left[\ d \text{ occurs in at least one list in } \bar{E}(d)\ \right] && (6.3)\\ &= 1 - P\left[\ d \text{ does not occur in any list in } \bar{E}(d)\ \right] && (6.4)\\ &= 1 - \prod_{i \in \bar{E}(d)} (1 - q_i(d)) && (6.5) \end{aligned}$$

assuming independence for tractability. Note that this independence assumption can be relaxed using a covariance-based technique as introduced in Subsection 6.2.3.

6.2.2 Combined Score Predictor & Selectivity Estimator

In the following, we write $A(d, Y')$ for the probabilistic event that d occurs in all lists Y' and in none of the remaining lists in $\bar{E}(d) \setminus Y'$ and $O(d, \bar{E}(d))$ for the probabilistic event that d occurs in at least one of the dimensions in $\bar{E}(d)$. Then the combined probability that a document d can get into the top-k results can be estimated as follows:

$$\begin{aligned} p(d) &:= P\left[\ d \in \textit{top-}k\ \right] && (6.6)\\ &= \sum_{Y' \subseteq \bar{E}(d)} P\left[\ A(d, Y') \wedge \sum_{i \in Y'} S_i > \textit{min-}k\ \right] && (6.7)\\ &\leq \sum_{Y' \subseteq \bar{E}(d)} P\left[\ A(d, Y') \wedge \sum_{i \in \bar{E}(d)} S_i > \textit{min-}k\ \right] && (6.8)\\ &= P\left[\ O(d, \bar{E}(d)) \wedge \sum_{i \in \bar{E}(d)} S_i > \textit{min-}k\ \right] && (6.9)\\ &= P\left[\ \sum_{i \in \bar{E}(d)} S_i > \textit{min-}k \mid O(d, \bar{E}(d))\ \right] \\ &\quad \cdot\ P\left[\ O(d, \bar{E}(d))\ \right] && (6.10)\\ &= p_S(d)\ \cdot\ q(d) && (6.11) \end{aligned}$$

This corresponds to a conjunctive combination of the probabilities from the score predictor and selectivity estimates, assuming independence between the score predictor and the selectivity estimator, however.

6.2.3 Feature Correlations

Assuming that documents occur independently in different lists may lead to a crude and often useless estimator as terms used in queries are frequently highly correlated. To capture this in our probability estimator, we precompute pairwise term covariances for terms in frequent queries. For two such terms and their corresponding lists L_i and L_j, we use a contingency table to capture co-occurrence statistics for these terms, e.g., using frequently used term pairs from query logs or by extracting 2-grams from a thesaurus.

In the following, we denote by l_i the length of list L_i and by l_{ij} the number of docs that are in both L_i and L_j. We then consider the random variable X_i which is 1 if some doc d is in L_i (the same distribution for all d, but not the same value, of course), and 0 otherwise. To predict $X_j(d)$ after knowing $X_i(d) = 1$, we have to compute the covariance $cov(X_i(d), X_j(d))$ of X_i and X_j. Following basic probability theory, we can estimate this covariance as

$$cov(X_i, X_j) = \frac{l_{ij}}{n} - \frac{l_i \cdot l_j}{n^2} \ . \tag{6.12}$$

In the remainder of this section, we show how feature correlations can be exploited for a better estimation of selectivities. We want to estimate the probability $q_i(d)$ that a document d occurs in the remainder of the list L_i given that it already has occurred in some lists $E(d)$, using the pairwise covariances of L_i with the lists in $E(d)$. First we consider the case where $E(d) = \{j\}$ consists of a single list. Using the equality

$$P[X_i \wedge X_j] = P[X_i] \cdot P[X_j] + cov(X_i, X_j) \tag{6.13}$$

for Bernoulli random variables, we can then derive

$$P[X_i|X_j] = \frac{P[X_i \wedge X_j]}{P[X_j]} \tag{6.14}$$

$$= \frac{P[X_i] \cdot P[X_j] + cov(X_i, X_j)}{P[X_j]} \tag{6.15}$$

$$= \frac{\frac{l_i}{n} \cdot \frac{l_j}{n} + \frac{l_{ij}}{n} - \frac{l_i \cdot l_j}{n^2}}{\frac{l_j}{n}} \tag{6.16}$$

$$= \frac{l_{ij}}{l_j} \tag{6.17}$$

$$\tag{6.18}$$

We would like to estimate $P[X_i = 1 \mid i \in E(d)] = P[X_i = 1 \mid X_1 = 1 \wedge X_2 = 1 \wedge \ldots \wedge X_j = 1]$ with $E(d) = \{1, 2, \ldots, j\}$ and the elements of $E(d)$ conveniently renumbered.

Since we only have pairwise covariance estimates, we work with the approximation $P[X_i = 1 \mid i \in E(d)] \geq \max_{j \in E(d)} P[X_i = 1 \mid X_j = 1]$ which yields

$$
\begin{aligned}
q_i(d) &:= P\,[\, X_i = 1 \mid i \in E(d)\,] && (6.19)\\
&\geq \max_{j \in E(d)} P[X_i = 1 \mid i \in E(d)] && (6.20)\\
&= \max_{j \in E(d)} \frac{l_{ij}}{l_j} && (6.21)\\
& && (6.22)
\end{aligned}
$$

Then we can now plug this correlation-aware estimation for the probability that a document occurs in a single list into the selectivity estimator from Section 6.2.1 and the combined score predictor from Section 6.2.2.

6.3 Sorted Access Scheduling

Recall that index lists are processed in batches of b sorted accesses. That is, the query engine fetches b index entries from all m query-relevant index lists, and these b entries can be distributed across the lists in an arbitrary manner. The priority queue Q for result candidates is rebuilt with updated priorities after each round of b such steps.

Our goal in sorted-access (SA) scheduling is to optimize the individual batch sizes b_i ($i = 1..m$) across all the lists, i.e., choose $b_1, \ldots, b_m$ so as to maximize some benefit function under the constraint $\sum_{i=1}^{m} b_i = b$. For the batched sorted access mode, the units of the scheduling decisions are entire batches or blocks of the inverted lists. In the following we will present our methods in terms of SAs to individual index entries; the block-oriented variant follows in a straightforward manner.

Inspired by the earlier work on simple scheduling heuristics [GBK01], our first method aims to reduce the scores at the index scan positions, the $high_i$ bounds, as quickly as possible. The rationale of this strategy is that low $high_i$ values result in lower *bestscores* of top-k candidates, which in turn enables us to prune more candidates more quickly. It turns out, however, that this strategy does not perform well in many cases. We have developed a more general and typically better performing scheduling strategy that considers an explicit notion of *benefit* of a candidate in Q and aggregates over all candidates for a judicious decision on the b_i steps. We will discuss the

score-reduction strategy in Section 6.3.1 and the benefit-aggregation strategy in Section 6.3.2. As we show in Appendix A.3.1, both strategies have a reduction to an NP-hard Knapsack problem, hence we have coined them `KSR` (Knapsack scheduling for Score Reduction) and `KBA` (Knapsack scheduling for Benefit Aggregation).

6.3.1 Knapsack Scheduling for Score Reduction

Given the current scan positions $pos_1, \ldots, pos_m$, the Knapsack Scheduling for Score Reduction (`KSR`) is looking for a schedule of $b_1, \ldots, b_m$ steps (with $b_1 + \ldots + b_m = b$), that maximizes the *total reduction* in bestscores of documents that are currently present in our candidate queue. For a candidate document $d \in Q$, $bestscore(d)$ reduces by

$$\Delta_i \;:=\; high_i - score_i(pos_i + b_i) \tag{6.23}$$

if $i \in \bar{E}(d)$, i.e., when we scan b_i elements further into list L_i and do not see the document d in the list L_i, and by 0 if $i \in E(d)$. Since the probability of seeing a particular document by scanning a small part of a list is close to zero, the expected reduction in $bestscore(d)$ can be considered as Δ_i as shown in Figure 6.1.

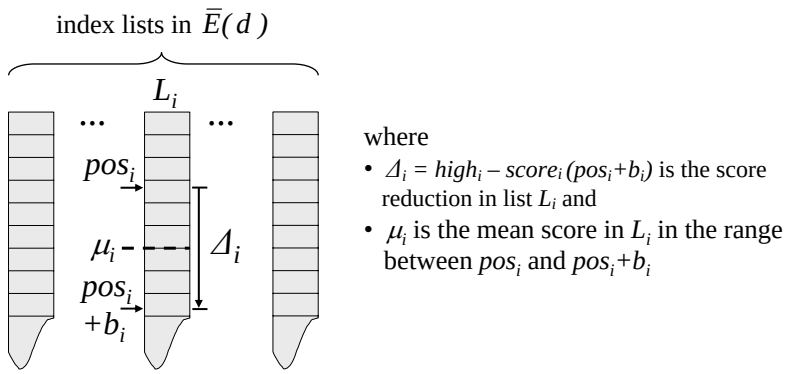


Figure 6.1: Expected score reduction and mean score for the next batch of b_i sorted accesses.

Hence the expected aggregated reduction in bestscores for all documents in Q is given by $w_i \cdot \Delta_i$ where $w_i := |\{d \in Q \mid i \in \bar{E}(d)\}|$ is the number of documents for which a reduction in bestscore is expected by scanning into list L_i. We can easily estimate the $score_i(pos_i + b_i)$ from the precomputed histograms, assuming a Uniform distribution of scores within each histogram

cell. Note that $score_i$ cannot fall below 0. We can now define our objective function for the choice of b_i values: maximize the score reduction

$$SR(b_1, \ldots, b_m) \quad = \quad \sum_{i=1}^{m} w_i \cdot \Delta_i \tag{6.24}$$

where we treat the Δ_i values as a (deterministic) function of the b_i choices (ignoring potential estimation errors caused by the histograms). Since candidates are assumed to be Uniformly distributed among all lists and remainder groups, the globally best score reduction should affect most partially evaluated candidates and yet unseen items and reduce their bestscores to the largest extent.

This problem is NP-hard, as we can reduce the well-known Knapsack problem to it; see Appendix A.3.1. However, usually the number of lists m for a query is small enough, and if we restrict our choice of b to a reasonably small number of batched index blocks (usually $c \cdot m$ for a constant c between 2 and 4), the number of possible schedules $(b_1, \ldots, b_m)$ for large list batches (e.g., multiples of the amount of tuples that fit on a disk sector) to compute the expected score reduction for all combinations is efficient enough to find their maximum without causing any noticeable overhead.

Greedy Heuristics Based on Score Gradients

Obviously, for arbitrary query sizes m and batch sizes b, we do not want to solve an NP-hard problem at query run-time, so we resort to efficient greedy heuristics for the Knapsack scheduling. For Knapsack itself, a simple technique is to compute utility/weight ratios and select items in greedy order of these values. The counterpart for scheduling is to consider the score gradients in the different lists as utility and the additional scan depth required to achieve a certain gradient as weight. To this end, we can compute the score look-ahead for each list if we made all b steps in that list, thus leading to:

$$gradient(L_i) \quad := \quad \frac{high_i - score_i(pos_i + b)}{b} \ . \tag{6.25}$$

These gradients reflect the relative value of the different lists. Again, the look-ahead score $score_i(pos_i+b)$ can easily be read off the precomputed index statistics. Then we may finally choose the actual b_i values in proportion to the gradients:

$$b_i \quad = \quad b \ \cdot \frac{gradient(L_i)}{\sum_{\nu=1}^{m} gradient(L_\nu)} \tag{6.26}$$

$$= \quad b \ \cdot \frac{\Delta_i}{\sum_{\nu=1}^{m} \Delta_\nu} \ . \tag{6.27}$$

6.3.2 Knapsack Scheduling for Benefit Aggregation

The Knapsack scheduling framework introduced in the previous subsection is intriguing and powerful, but it solely aims at reducing the local scores at the current scan positions as quickly as possible which might not be the best optimization criterion, because it does not exhaust any available candidate statistics from the queue and the current top-k list. Although it allows us to identify some low-scoring candidates and prune them earlier, it does not necessarily lead to more information about the high-scoring candidates. In particular, we may not find any additional scores of the current top-k results, so that we cannot improve the *min-k* threshold, which would be another way of pruning many candidates quickly. Key to increasing the *min-k* threshold would rather be to perform additional random accesses; we will come back to this issue in the next section.

An aspect directly related to SA scheduling is that we do not only want to reduce the bestscore bounds of some candidates as much as possible, but are actually more concerned about the bestscore bounds of those candidates that are close to the *min-k* threshold. More generally, we would prefer a modest bestscore reduction of many candidates over a big reduction for some smaller fraction only.

To address these issues, the Knapsack Scheduling for Benefit Aggregation (`KBR`) defines an explicit formalization of the *benefit* that we obtain from scanning forward by $(b_1, \ldots, b_m)$ positions in the m index lists, taking into consideration not only the current scan positions and score statistics, but also the knowledge that we have compiled about the documents seen so far during the scans. Benefit will be defined for each document, and we will then aggregate the benefits of all documents in the current top-k or the candidate queue Q. Observe that if a candidate document d has already been seen in list L_i, then neither $bestscore(d)$ nor $worstscore(d)$ changes when we scan L_i further. So, for each list L_i, we will consider only the documents $d \in Q$ which are have not been seen in L_i yet, i.e., $i \in \bar{E}(d)$. Then the probability $q_i^{b_i}(d)$ of seeing d in L_i in the next b_i steps is

$$
\begin{aligned}
q_i^{b_i}(d) &:= P\left[\ d \text{ in next } b_i \text{ elements of } L_i \mid i \in \bar{E}(d)\ \right] && (6.28)\\
&= P\left[\ d \text{ in next } b_i \mid d \in L_i \wedge i \in \bar{E}(d)\ \right] \\
&\quad \cdot P\left[\ d \in L_i \mid i \in \bar{E}(d)\ \right] && (6.29)\\
&= \frac{b_i}{l_i - pos_i} \cdot P\left[d \in L_i \mid i \in \bar{E}(d)\right] && (6.30)\\
&= \frac{b_i}{l_i - pos_i} \cdot P\left[X_i = 1 \mid i \in \bar{E}(d)\right] && (6.31)
\end{aligned}
$$

$$\leq \quad \frac{b_i}{l_i - pos_i} \cdot \max_{j \in E(d)} \frac{l_{ij}}{l_j} \tag{6.32}$$

We can estimate the reduction in $bestscore(d)$ of a candidate document $d \in Q$ with regard to list L_i as $(1 - q_i^{b_i}(d)) \cdot \Delta_i$. On the other hand, if a document d is actually found in L_i by scanning further to depth b_i, the $worstscore(d)$ of d increases which in turn contributes to increasing *min-k* and, thus, in pruning more documents. Figure 6.1 also shows that the expected *gain* in $worstscore(d)$, when list L_i is scanned further to depth b_i, is given by $q_i^{b_i}(d) \cdot \mu(pos_i, b_i)$, where $\mu(pos_i, b_i)$ is the mean score of the documents from current scan position pos_i to $pos_i + b_i$. We can estimate $\mu(pos_i, b_i)$ as well from the precomputed histogram.

Now we can define our benefit function for every candidate document $d \in Q$ not already seen in list L_i as

$$\mathrm{Ben}_i(d, b_i) \quad = \quad \left(1 - q_i^{b_i}(d)\right) \cdot \Delta_i \; + \; q_i^{b_i}(d) \cdot \mu(pos_i, b_i) \tag{6.33}$$

and the total benefit of scanning to depth b_i in L_i as

$$\mathrm{Ben}_i(b_i) \quad = \quad \sum_{\substack{d \in Q \\ i \in \bar{E}(d)}} \mathrm{Ben}_i(d, b_i) \tag{6.34}$$

Finally, we can define the overall benefit for a schedule $s = (b_1, \ldots, b_m)$ by a simple benefit aggregation:

$$\mathrm{Ben}(s) \quad = \quad \sum_{i=1}^{m} \mathrm{Ben}_i(b_i) \tag{6.35}$$

So we are looking for a schedule s for which the benefit $\mathrm{Ben}(s)$ is maximized. This notion of an overall benefit includes an implicit weighting of lists, by giving higher weight to the lists for which we have many documents in the queue that have not yet been seen there and which would benefit from a significant reduction of the $high_i$ bounds for these lists. Thus scanning on these lists could make the decisive difference between pruning many candidates or having to keep them in the queue. Again, the NP-hardness of this problem can be proven by a similar construction as sketched in Appendix A.3.1. With the above definitions, our goal clearly is to maximize the overall benefit $Ben(s)$ subject to the constraint $b_1 + \ldots + b_m = b$. As justified in the previous subsection, we can usually compute the optimal solution exactly without any noticeable overhead, when we allow only a few possible combinations of large, discrete batching depths b_i in each round, and the number of input lists m is small.

Greedy Heuristics Based on Aggregated Benefits

Just like in the previous subsection, this is a Knapsack-related NP-hard problem. So even if we had all $Ben_i(b_i)$ values precomputed and stored in a table for all possible choices of $b_1, \ldots, b_m$, we would not want to solve this optimization problem at query runtime exactly for the general case. Analogously to the approach of the previous subsection, we may rather employ a fast and greedy approximation.

We only consider the $Ben_i(b)$ values that reflect the relative importance of the different lists (in the current state of the query processing) for a constant batch size b and with regard to all candidates currently being in the queue. Finally, we may choose the b_i values in proportion to the $Ben_i(b)$ importance levels of the various lists:

$$b_i = b \cdot \frac{Ben_i(b)}{\sum_{\nu=1}^{m} Ben_\nu(b)} \tag{6.36}$$

6.4 Random Access Scheduling

Random access (RA) scheduling is crucial both in the early and the late stages of top-k query processing. In the early stage, it is important to ensure that the *min-k* threshold moves up quickly so as to make the candidate pruning more effective as the scans proceed and collect large amounts of candidates. Later, it is important to avoid that the algorithm cannot terminate merely because of a few pieces of information missing about a few borderline candidates. In the following, we present various strategies for deciding when to issue RAs and for which candidates in which lists. Some of them have a surprisingly simple heuristic nature, others are cost-model-driven. All of them and also the hybrid methods can be easily integrated with different strategies for SA scheduling. Following the literature [CwH02, MBG04], we refer to score lookups by RAs as *probing*.

6.4.1 Last-Probing

The *Last-Probing* approach works in two strictly separated phases: the first phase performs only SAs and no RAs at all, then the algorithm stops the index scans and is completed by the second phase with all RAs that are necessary to identify the final top-k result. Obviously, the second phase can start only when $\sum_{i=1}^{m} high_i \leq min\text{-}k$, but this point is usually reached fairly soon in real queries on real data. With this constraint, the main criterion for switching from the first to the second phase is based on the accumulated costs for SAs in phase 1 so far and the expected costs for RAs in phase 2.

It is derived from a simple cost model that reflects the typical behavior of TA-style top-k query processing.

Suppose we want to move from phase 1 to phase 2 at point x (where x can be in terms of discrete time or in terms of scan depth or number of SAs). The number of SAs up to this point is typically linear, hence the cost is

$$C_{SA}(x) \quad = \quad c_S \cdot ax \tag{6.37}$$

with some coefficient a and c_S the cost for a single SA. The number of RAs in phase 2, starting at point x, should go down with increasing x. Typically, the cost would decrease much faster than linear, for example, in the form

$$C_{RA}(x) \quad = \quad c_r \cdot \frac{1}{cx} \tag{6.38}$$

with some coefficient c and c_R the cost for a single RA as shown in Figure 6.2. This assumption is invigorated by the observation that after the first batches of sorted accesses, namely when $\sum_i^m high_i \leq min\text{-}k$, a vast majority of candidates is immediately pruned and then only very few candidates are that are either going into the final top-k or very close matches are kept in the queue for a long time until they would finally be uncovered through sorted accesses alone. This is the typical behavior (and the weakness) of the NRA algorithm. So the RA costs to resolve the remaining candidates in the queue decreases at a much higher rate than the SA costs are increasing which would be the case when we decided to keep on scanning sequentially. Similarly, the chances to hit a final top-k item among the queued candidates and thus increase $min\text{-}k$ (which leads to immediate pruning of more candidates increases with only a small amount of candidates left (see Section 6.4.1).

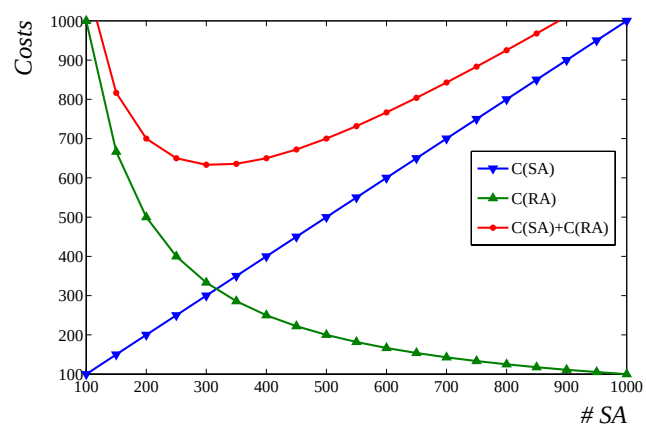


Figure 6.2: Expected trend of the C_{SA} and C_{RA} costs.

Then, minimizing the total cost

$$C(x) = C_{SA}(x) + C_{RA}(x) \tag{6.39}$$

$$= c_S \cdot ax + c_R \cdot \frac{1}{cx} \tag{6.40}$$

would choose the optimal point x. The solution (easily found by differentiation) is

$$x_{opt} = \sqrt{\frac{c_R}{c_S} \cdot \frac{1}{ac}} \ . \tag{6.41}$$

For this point, the costs

$$C_{SA}(x_{opt}) = C_{RA}(x_{opt}) \tag{6.42}$$

$$= \sqrt{c_R \cdot c_S \cdot \frac{a}{c}} \tag{6.43}$$

become equal regardless of the values for the coefficients a and c and the ratio c_R/c_S of the costs. That is, the optimal switching point (under this simple cost model) is when the expected RA cost is the same as the SA cost consumed so far.

The SA cost consumed so far is trivially computed by simply counting the index-scan steps; and the remaining RA cost is also easy to determine from the candidate queue and the $\bar{E}$ sets:

$$C_{RA} = \frac{c_R}{c_S} \cdot \sum_{d \in Q} |\bar{E}(d)| \tag{6.44}$$

Thus, we are normalizing the cost of an SA to 1 and weighting RAs according to the ratio c_R/c_S; then the actual values for c_R and c_S do not make any difference for the scheduling decisions, just the ratio.

Estimating the Number of Remaining RAs

According to the `Last` heuristic, we stop performing sorted accesses once the number of candidate documents $|Q|$ becomes less than a fraction of c_S/c_R of the total number of sorted accesses done until that point. This guarantees that the total random access cost is never more than the total sorted access cost. Now the number of random accesses that have do be done in the last round is often (but not necessarily) much less than the size of the queue. This is, because certain random lookups might increase the scores of the respective documents to such an extent, that they already get promoted into the top-k, and thereby increase the *min-k* score which may in turn lead to an

increased pruning of candidates from the queue. In this section, we develop an accurate *estimator* for this number of actually remaining random accesses.

Let us consider the queue after some round and assume an ordering of the documents by descending bestscores, i.e., highest bestscore first. For the ith document in that ordering, let W_i and B_i denote its worstscore and bestscore, respectively, and by F_i its final score (which we do not know before doing random accesses, unless $W_i = B_i$). Now consider the lth document (in the bestscore ordering), and let k' be the number of top-k items with worstscore below B_l. Then it is not hard to see that there will be a random lookup for this l^{th} document, if and only if at most k' of the $l-1$ documents $d_1, \ldots, d_{l-1}$ have a final score larger than B_l. Let R_l be the random indicator variable that is 1 if that happens and 0 otherwise. Let $p_{i,l} := P[F_i > B_l]$, which can be computed as described in Section 6.2.2. Since the $p_{i,l}$ are small, and l tends to be large, the number of i for which $F_i > B_l$ can be approximated very accurately by a random variable X_l with a Poisson distribution with mean $p_{1,l} + \cdots + p_{l-1,l}$. We then have

$$E(R_l) \quad = \quad P[R_l = 1] = P[X_l < k] \tag{6.45}$$

which can be computed very efficiently and accurately by means of the Incomplete Gamma function [PFTV92].

As described so far, the probabilities $p_{1,l}, \ldots, p_{l-1,l}$ would have to be computed from scratch for every document. The time for computing $\sum_l E(R_l)$ as an estimate for the number of random accesses would then be quadratic in the number of documents in the queue. We improve on this by approximating $p_{i,l} = P[F_i > B_l]$ by

$$\tilde{p}_{i,l} \quad = \quad P\left[F_i > \textit{min-k}\right] \quad \cdot \quad \frac{B_l - \textit{min-k}}{B_i - \textit{min-k}} \tag{6.46}$$

Note that by the bestscore ordering we have that $B_l \leq B_i$, for $i < l$. It then suffices to compute $P[F_i > \textit{min-k}]$, once for each document i, and to maintain, while processing the documents in order of descending bestscores, the number of top-k items which are smaller than the current document, which can be done in linear overall time. It is not hard to see, that from these quantities, $\sum_{i=1}^{l-1} \tilde{p}_{i,l}$ can be computed in constant time, for any given l. Similarly to original CA baseline, the RAs are then scheduled in descending order of $\textit{bestscore}(d)$, denoted as `Last-Best`.

6.4.2 Ben-Probing

The beneficial probing strategy, *Ben-Probing* for short, extends the Last-Probing by a probabilistic cost model for assessing the benefit of making

RAs to a number of the most promising candidates versus continuing with SAs in the index scans. The cost comparison is updated periodically every b steps, i.e., whenever we need to make the next SA-batching decision anyway. The cost is computed for each document d_j in the candidate queue or the current top-k separately; obviously SA costs per document are then fractions of the full SA costs as the index scan steps are amortized over multiple documents. Then we can either schedule RAs for individual documents based on the outcome of the cost comparison, or we can batch RAs for multiple candidates and would then simply aggregate the per-candidate RA costs. In the following we first develop the cost estimates, and then come back to the issue of specific scheduling decisions.

For both cost categories, we consider the *expected wasted cost (EWC)* which is the expected cost of random (or sorted) accesses that our decision would incur but would not be made by an optimal schedule that would make random lookups only for the final top-k and traverse index lists with minimal depths. To compute the EWCs, we set the cost of an SA to 1 and the cost of an RA to c_R/c_S, hence the model uses only the cost ratio, not the actual costs.

For looking up unknown scores of a candidate document d in the index lists $\bar{E}(d)$, we would incur $|\bar{E}(d)|$ random accesses which are wasted if d does not qualify for the final top-k result. We can compute this probability using the combined score estimator from Section 6.2.2 and exploiting correlations as shown in Section 6.2.3, as

$$
\begin{aligned}
P\,[\; d \notin \textit{top-}k \;] &= 1 - p(d) && (6.47)\\
&= 1 - p_S(d) \cdot q(d) && (6.48)\\
&= 1 - p_S(d) \cdot \left(1 \; - \prod_{i \in \bar{E}(d)} (1 - q_i(d)) \right) && (6.49)\\
&\leq 1 - p_S(d) \cdot \left(1 \; - \prod_{i \in \bar{E}(d)} \left(1 - \max_{j \in E(d)} \frac{l_{ij}}{l_j} \right) \right) && (6.50)
\end{aligned}
$$

Then the random accesses to resolve the missing scores have expected wasted cost:

$$
EWC_{RA}(d) \;:=\; |\bar{E}(d)| \;\cdot\; (1 - p(d)) \;\cdot\; \frac{c_R}{c_S} \qquad (6.51)
$$

Analogously, the next batch of b sorted accesses for an additional depth b_i at index list L_i, with $\sum_i b_i = b$, incurs a fractional cost to each candidate in the priority queue, and these total costs are shared by all $|Q|$ candidates.

For a candidate d, the sorted accesses are wasted, if either we do not learn any new information about the total score of d (that is, when we do not encounter d in any of the m remainder dimensions), or if we encounter d, but it does not make it to the top-k. Denoting the probability of seeing d in the ith list in the next b_i steps as $q_i^{b_i}(d)$ like in Section 6.3.2, we can compute the probability $q^b(d)$ of seeing d in at least one list in the batch of size b as

$$q^b(d) := 1 - P[\ d \text{ not seen in any list }] \tag{6.52}$$

$$= 1 - \prod_{i \in \bar{E}(d)} (1 - P[\ d \text{ seen in } L_i \text{ in next } b_i \text{ steps }]) \tag{6.53}$$

$$= 1 - \prod_{i \in \bar{E}(d)} \left(1 - q_i^{b_i}(d)\right) \ . \tag{6.54}$$

$$\tag{6.55}$$

Hence the probability of *not* seeing d in any list is $1 - q^b(d)$. The probability that d is seen in at least one list, but does not make it into the final top-k results can be computed as

$$q_S(d) := (1 - p_S(d)) \cdot q^b(d) \tag{6.56}$$

in analogy to Section 6.2.2. Then the total costs for the next batch of b sorted accesses are shared by all candidates in Q, and this incurs expected wasted cost:

$$EWC_{SA} := \frac{b}{|Q|} \cdot \sum_{d \in Q} \left((1 - q^b(d)) + (1 - p_S(d)) \cdot q^b(d)\right) \tag{6.57}$$

$$= \frac{b}{|Q|} \cdot \sum_{d \in Q} \left(1 - p_S(d) \cdot q^b(d)\right) \tag{6.58}$$

We can now replace the real costs (as counted in the *Last-Probing*) with the expected wasted costs EWC_{RA} and EWC_{SA} for the Ben-Probing. In order to trigger random accesses for specific candidates, we always consider the *cumulated* EWC_{RA} costs and compare them to the *cumulated* EWC_{SA} of all batches done so far. For `Last-Ben`, we exclusively perform SA batches until the sum of the expected wasted costs of *all* remaining candidates in the queue is less than the cumulated expected wasted costs of all previous SA batches; we then perform the RAs for all documents in the queue in ascending order of the candidates' EWC_{RA}.

For each candidate d, we actually perform the RAs one at a time in ascending order of the remaining index list selectivity $(l_i - pos_i)/n$ for all $i \in \bar{E}(d)$ (because if d fails the final top-k, it will probably fail at a lowly selective

list first), thus counting a single RA for each candidate and list. After each random access, it is tested whether the candidate document can qualify for the top-k results; if the candidate can be dismissed, i.e., $bestscore(d_j) \leq$ *min-k*, all subsequent random accesses are canceled, and we may safe some RA costs for another candidate this way. If otherwise $worstscore(d_j) >$ *min-k*, the candidate is promoted into the (intermediate) top-k list. In any case, after triggering the batch of random accesses for any candidate d_j in the queue, d_j will be discarded from the queue and is either dropped "forever" or made sure to belong to the top-k results.

The cost comparisons and scheduling decisions are made only once at the start of the entire sequence of random accesses for a candidate. The cost comparisons have a fairly low overhead and are performed whenever the priority queue is rebuilt. As an additional variant, one could perform the cost comparisons for each of the top-ranked (≈ 100) candidates sorted by their bestscores only.

Chapter 7

Dynamic & Self-tuning Query Expansion

Query expansion is an indispensable technique for evaluating difficult queries where good recall is a problem. Examples of such queries are the ones in the TREC Robust track [TRE], e.g., queries for "transportation tunnel disasters" or "ship losses" on the Acquaint news corpus. State-of-the-art approaches use one or a combination of the following sources to generate additional query terms: thesauri such as WordNet with concept relationships and some form of similarity measures, explicit user feedback or pseudo relevance feedback, query associations derived from query logs, document summaries such as Google top-10 snippets, or other sources of term correlations. In all cases, the additional expansion terms are chosen based on similarity, correlation, or relative entropy measures and a corresponding threshold. For difficult retrieval tasks like the above, query expansion can improve both precision at the top ranks, as well as recall over the complete retrieval result and, hence, typical IR benchmark metrics such as Mean Average Precision (MAP) by a significant margin (see, e.g., [Kwo04, LLYM04]).

Following [BZ04a, BZ04b], in contrast to a mere benchmark setting such as TREC, applying these techniques in a real application with unpredictable ad-hoc queries, e.g., in digital libraries, intranet search, or Web communities, faces three major problems:

1) The threshold for selecting expansion terms needs to be carefully hand-tuned, and this is highly dependent on the application's corpus and query workload.

2) An inappropriate choice of the sensitive expansion threshold may result in either not achieving the desired improvement in recall (if the

threshold is set too conservatively) or in high danger of topic dilution (if the query is expanded too aggressively).

3) The expansion may often result in queries with more than 50 or 100 terms, which in turn leads to very high computational costs in evaluating the expanded queries over inverted index lists in a non-conjunctive, i.e., "andish", manner.

TopX addresses the above three issues and provides a practically viable, novel solution. Our key techniques for making query expansion efficient, scalable, and self-tuning are to avoid aggregating scores for multiple expansion terms of the same original query term and to avoid scanning the index lists for all expansion terms. For example, when the term "disaster" in the query "transportation tunnel disaster" is expanded into "fire", "earthquake", "flood", etc., we do not count occurrences of several of these terms as additional evidence of relevance. Rather than that, we use a score aggregation function that counts only the *best match* of a document for a given set of expansion terms of the same original query term, optionally weighted by the similarity (or correlation) of the expansion term to the original term. Furthermore and most importantly for efficiency, we open scans on the index lists for expansion terms as late as possible, namely, only when the best possible candidate document from a list can achieve a score contribution that is higher than the score contributions from the original term's list at the current scan position or any list of expansion terms with ongoing scans at their current positions. The algorithm conceptually merges the index lists of the expansion terms with the list of the original query term in an incremental, on-demand manner during the runtime of the query. For further speed-up, probabilistic score estimations can be used, thus leveraging our previous work on convolutions for score distributions, different index lists selectivities, and feature correlations (see Sections 5.2 and 6.2).

7.1 Static vs. Dynamic Query Expansion

7.1.1 Static Expansions & Topic Drifts

Traditional query expansion methods select expansion terms whose thematic similarity to the original query terms is above some specified threshold, e.g., using the Rocchio [Roc71] method or Robertson and Sparck-Jones [RJ76] weights, thus generating an "andish" query with much higher dimensionality.

Figure 7.1 shows a typical expansion scenario for the query "transportation tunnel disasters" which is drawn from the TREC 2004 Robust track

and explicitly marked by TREC as a particularly "hard" query based on retrieval results from previous TREC ad hoc tracks and tasks. The main reason

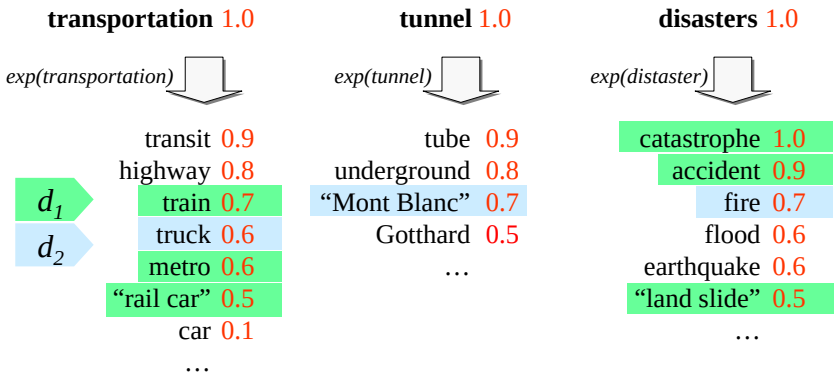


Figure 7.1: Topic drift through overly aggressive static query expansions.

why standard IR systems fail in this case is that the given corpus, namely the Aquaint News corpus with news articles from the Financial Times, LA Times, etc. (see Section 9.2), is very unlikely to contain many documents with this particular combination of keywords (this might not be the case for a different corpus such as a large Web crawl, however). Most notably, the original query does not specify any particular instance of such an incident which would be more likely to be found on the Aquaint corpus, but rather yields a vague description of the topic in general. In such a case, query expansion offers the potential not only to increase recall but also precision at the upper ranks which makes the aforementioned expansion techniques attractive for a top-k engine with k being in the order of 10-100, because, with the original keyword query, we virtually start with nothing.

Topic Drifts

Let us for now suppose we have a good source for expansion terms and some notion of similarity between the original topic term and the expanded term or phrase concept, and each of the three keywords has been expanded into the three expansion groups with respective similarities as shown in Figure 7.1. An initial query using all keywords as a plain set of expansion terms might identify the two documents d_1 and d_2 as shown in Figure 7.1. We see that d_2 contains some matches for expanded terms from *all* the three original query concepts; whereas d_1 contains much more matches overall for expansions from "transportation" and "disaster", but not a single match to the "tunnel" concept. With plain summation being used for score aggregation, d_1 is very likely to be ranked much higher than d_2 which would in turn be much more

likely to be relevant to the original topic than d_1. This is exactly what we call a *topic drift*.

Note that these situations are hard to foresee, because term correlations in queries often diverge from term correlations we observe among the documents in the corpus. Typically, there is no common generative model from which queries, expansion terms (e.g., a large common sense thesaurus), and documents can be drawn. This is also the reason why certain entropy-based IR methods such as the Divergence From Randomness (DFR) model [AvR02, ACR03] work well for many corpora and queries; often the most relevant documents for a query are those documents whose document-specific term distribution *diverges* from the collection-specific term distribution to the largest extent.

Note that the emphasis of this thesis is not on the best possible choice of relevant expansion terms; we rather aim at providing the algorithmic basis for an efficient and robust, top-k-style query processing for any given, large-scale expansion.

7.1.2 Dynamic & Incremental Query Expansion

The novel approach presented here addresses all three problems described in Section 7.1 by *dynamically* and *incrementally merging* the inverted lists for the potential expansion terms with the lists for the original query terms. We introduce a novel notion of *best match* score aggregation that only allows for the best match per expansion group to contribute to the final document score, thus reflecting the semantic structure of the query directly in the query precessing and score aggregation. The algorithm is implemented as an Incremental Merge operator that can be embedded into other pipelined and non-blocking Incremental Merge or nested top-k operators. This approach even allows for a special case of Boolean retrieval, where conjunctive query evaluation is enforced at the top-level operators and each expansion group is evaluated in a disjunctive manner.

Up to this point, sorted accesses to inverted lists have been performed on static, fully precomputed and materialized lists, where each list directly corresponds to a keyword condition denoted by a query. In this chapter, we generalize the notion of inverted lists toward *dynamic views* over a set of physically stored lists that are incrementally merged on demand, with minimum overhead in the merging algorithm itself as illustrated in Figure 7.2. Let us suppose we want to expand the term "disasters" in our example query "transportation tunnel disasters". Where a static expansion would just add expansion terms like "fire" or "accident" to the original query, optionally weighted by some similarity score, our approach utilizes a specialized opera-

tor that exploits the explicit structure defined by the original query and its expansions. Our approach incrementally merges the inverted lists associated with each expansion set into a virtual index list that provides sorted access to the outside top-k operator in descending order of combined similarity score sim and local scores $s_i(d)$. Note that the original query term"disaster" is kept in the expanded query, too; typically with a perfect similarity score of 1.

The great advantage of this method is that we do not have to predetermine the depth of the merging process itself. The superordinate top-k operator just incrementally inquires the subordinate Incremental Merge operator for the next document, thus preserving the efficient sorted access paradigm.

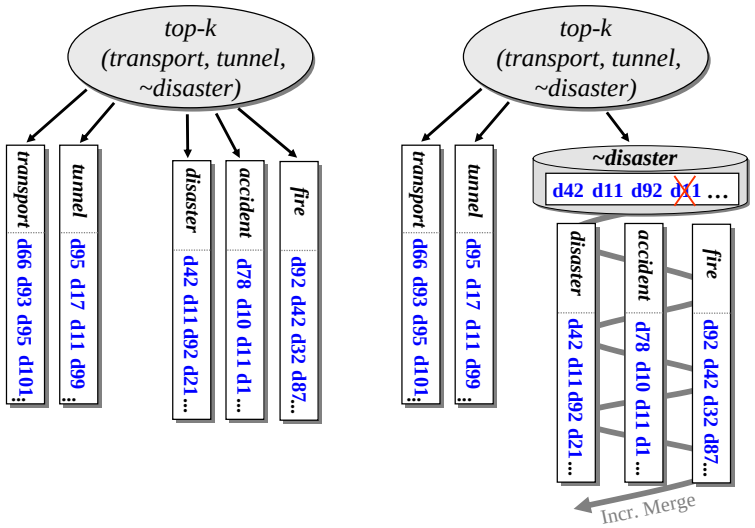


Figure 7.2: Static query expansions (left) vs. dynamic expansions (right), both embedded into a top-k query processor.

7.2 Thesaurus-based Query Expansion

We generate potential expansion terms for queries using a thesaurus database based on WordNet [Fel98]. WordNet is the largest electronically available, common-sense thesaurus with more than 120,000 semantic concepts, consisting of single terms and as well as explicitly identified phrases, and more than 300,000 handcrafted links that define the way how the concepts or *synsets* (i.e., sets of synonyms that refer to the same meaning) in the WordNet graph are related. The basic structure of WordNet with regard to the hypernym relationship only is "mostly" a tree structure which is the reason why WordNet is often referred to as a hierarchical thesaurus (HT) which is not exactly

true, since multiple inheritance is allowed and some concepts with multiple hypernyms break this tree structure. Taking all the 16 distinct edge types or concept relationships into account, WordNet forms a large concept graph as cycles are also possible.

Query expansion techniques used in IR typically suffer from the following two common phenomena of word usage in natural language:

1) *Polysemy:* A term can have different meanings depending on the context that it is used in.

2) *Synonymy:* Multiple terms have the same meaning; together with 1) the situation may become mutually context sensitive.

In order to address these problems, a query term t is mapped onto a WordNet concept c by comparing some form of textual context of the query term (i.e., the description of the query topic or the summaries of the top-10 results of the original query when relevance feedback is available) against the context of synsets and glosses (i.e., short descriptions) of possible matches for c and its neighbors in the ontology graph. The mapping uses a simple form of *Word Sense Disambiguation* (WSD) by choosing the concept with the highest similarity of each two context pairs.

7.2.1 Word Sense Disambiguation

Word Sense Disambiguation (WSD) has a long tradition and is an active research issue in the field of computational linguistics. We focus on so-called *unsupervised WSD* methods, i.e., fully automatic WSD. In the following we briefly discuss two unsupervised WSD approaches that have been developed and conducted in the context of our work on document classification and query expansion [TSW03, MTV+05].

As an example for our efforts, consider the the term "goal" which yields the following different word senses when queried in WordNet:

1) *{goal, end,...}* – the state of affairs that a plan is intended to achieve and that (when achieved) terminates behavior to achieve it; "the ends justify the means"

2) *{goal}* – a successful attempt at scoring; "the winning goal came with less than a minute left to play"

and two further senses. By looking up the synonyms of these word senses, we can construct the synsets *{goal, end, content, cognitive content, mental object}* and *{goal, score}* for the first and second meaning, respectively.

As each of the meanings is connected to different concepts in the ontology graph, a reliable disambiguation and choice of the seed concepts is a crucial precondition for any subsequent expansion or classification technique.

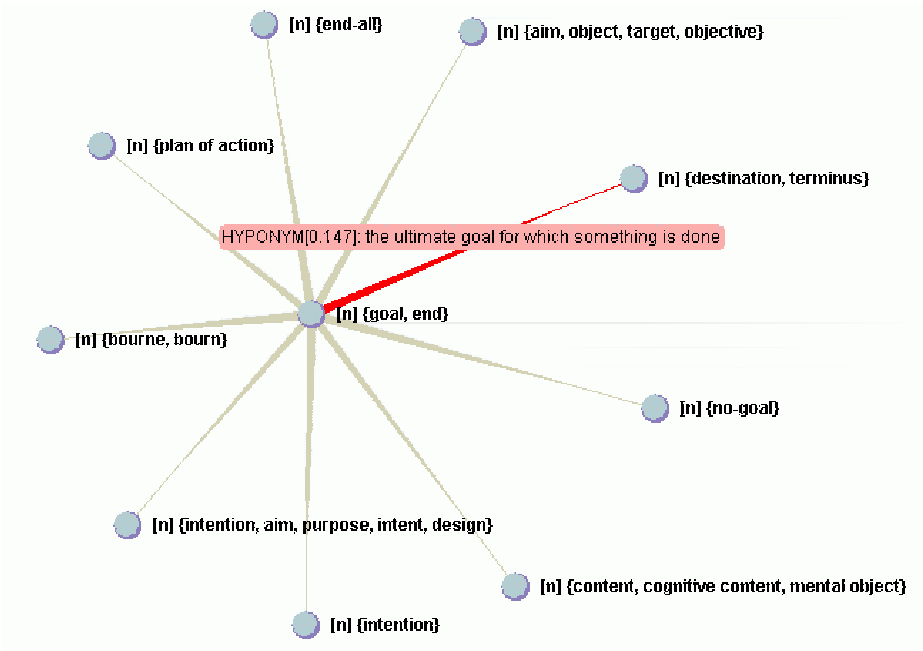


Figure 7.3: Visualization of the concept neighborhood graph for one possible meaning of the word 'goal'.

Now the key question is of course: which of the possible senses of a word is the right one? Our approach to answer this question is based on word statistics for some local context of both the term that appears in a part of a document or a keyword query and the candidate senses that are extracted from the concept graph.

Independent Mapping

In [TSW03], we coined the first approach the *Independent Mapping* or *Independent Disambiguation*, because each term or n-gram out of a given text sequence (e.g., a short keyword query or a paragraph in a document) is mapped individually onto its most likely meaning in the concept graph without taking the mapping of the word sequence "as a whole" into account (considering also the relationships between the mappings of these terms or n-grams).

In order to identify the largest possible subsequences of n-grams out of a given sequence, let us first consider a word sequence $w_1, ..., w_m$. Starting with the first word w_1 at position $i = 1$ in the sequence and a small lookahead

distance m' of at most 5 words, we use a simple window parsing technique to determine the largest subsequence of words that can be matched with a phrase contained in WordNet's synsets to identify an initial set of possible word senses $s_{i_1}, \ldots, s_{i_p}$. If we have successfully matched the current sequence $w_i, \ldots, w_{i+m'}$, we increment i by m' and continue the mapping procedure on the suffix of the sequence; if we could not match the current sequence onto any phrase denoted by a WordNet concept, we decrement m' by 1 and try the lookup again until $m' = 1$. After performing that subroutine, i is again incremented by 1 until $i = m$.

Fortunately, phrases of length 2 or 3 hardly ever exhibit more than one distinct meaning in WordNet, whereas in fact most single keywords match more than one semantic concept and, thus, are highly ambiguous. If the word sequence consists of some piece of natural language, we can improve the accuracy of this mapping procedure by using a so called Part-of-Speech (POS) grammatical tagger that annotates the words prior to disambiguation with their role in a sentence (e.g., verb, noun, adjective, etc.). The simple word "run", for example, has 57 distinct meanings in WordNet, out of which 16 are marked as nouns and 41 are verbs.

For a given term or n-gram t, we consider either the query that it occurred in, or, for a natural language document, a short context of some sentences that enclose that term, since the meaning of a term may of course vary across a large document. For an XML element this may be the full-content text its embedded descendant nodes. We denote this as the *local context* $con(t)$ *of* t. For a candidate word sense s, we extract synonyms, all immediate hyponyms and hypernyms, and also the hyponyms of the hypernyms (i.e., the siblings of s in the HT). Each of these has a synset and also a short explanatory text, coined "gloss" in the WordNet terminology. We form the union of the synsets and corresponding glosses, thus constructing a *local context* $con(s)$ *of sense* s extracting also n-grams from synsets and glosses. As an example, the context of sense 1 of the word "goal" (see Figure 7.3) corresponds to the bag of words *{goal, end, state, affairs, plan, intend, achieve, ..., content, cognitive content, mental object, perceived, discovered, learned, ..., aim, object, objective, target, goal, intended, attained, ... }*, whereas sense 2 would be expanded into *{goal, successful, attempt, scoring, winning, goal, minute, play, ..., score, act, game, sport, ... }*.

The final step toward disambiguating the mapping of a term onto a word sense is to compare the term context $con(t)$ with the context of candidate concepts $con(s_1)$ through $con(s_p)$ in terms of a similarity measure between two bags of words. The standard IR measure for this purpose would be the cosine similarity between $con(t)$ and $con(s_j)$, or alternatively the Kullback-Leibler divergence [BYRN99] between the two word frequency distributions

(note that the context construction may add the same word multiple times, and this information is kept in the word bag). Our implementation uses the cosine similarity between the TF·IDF vectors of $con(t)$ and $con(s_j)$ for its simpler computation.

Finally, we map term t onto that sense s_j whose context has the highest similarity, i.e., the lowest cosine distance, to $con(t)$. We denote this word sense as $sense(t)$. If there is no overlap at all, e.g., if the context denoted by a keyword query consists only of a single term, namely the one that is about to be expanded, we choose the sense that has the highest a-priory probability, i.e., the one with the lowest IDF-value.

Compactness-based Disambiguation

Another intriguing option of performing highly accurate, non-supervised WSD is the compactness-based disambiguation method that has been initially sketched by Vazirgiannis et al. in [MTV04] and recently refined in [MTV+05]. Unlike the Independent Mapping, this method aims at mapping a *whole set* of terms or n-grams, e.g., extracted from a natural language sentence, onto their most compact representation in the concept graph in a single, holistic step. The compactness-based WSD algorithm is based on the intuition that adjacent terms extracted from a piece of text are expected to be semantically close to each other. Given a set of adjacent terms, the disambiguation algorithm considers all the candidate sets of senses and outputs the set of senses that exhibits the highest level of semantic relatedness. Therefore, the main component of our WSD algorithm is the definition of a semantic compactness measure for sets of senses.

The compactness measure utilized in here is defined as follows:

Definition 7.2.1 (Semantic Compactness) *Given a concept graph $G = (N, V)$ with nodes N and vertices $V \subseteq N \times N$ and a set of senses $S = (s_1, ..., s_n)$ with $s_i \in N$, the compactness of S is defined as the cost of the Steiner Tree [CLRC01] over V that contains all $s_i \in S$.*

Note that the problem of computing the Steiner Tree over arbitrary graphs to get the compactness for a given mapping of terms onto possible concepts is already NP-complete. Another issue, potentially adding excessive computational load, is the large number of combinations of possible mappings for terms onto their senses, when each term is highly ambiguous (with 5 or more possible senses) and a term set of large cardinality is considered for disambiguation. In order to address this combinatorial complexity, we reduce the search space by including n-grams whenever possible using the window parsing technique described for the Independent Disambiguation approach

and restricting the mapping step to fairly small sets only, e.g., concept candidates extracted from a single sentence or a short keyword query.

Figure 7.4 illustrates the compactness-based disambiguation of the term "wind" with regard to two different contexts. The first pair, *{wind, thunder}*, finds a compact Steiner Tree mapping for "wind" and "thunder" as two phenomena of "weather". The second pair, *{wind, guitar}*, can be compactly mapped under "music instruments". Hence, the ambiguous term "wind" can successfully be assigned two different meanings depending on the context it occurred in. Note that both concept contexts would yield totally different expansion terms.

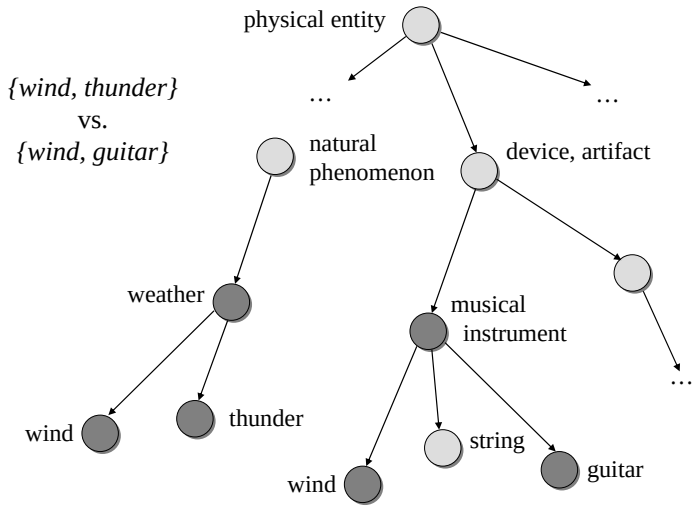


Figure 7.4: Compactness-based disambiguation of two term pairs.

The two unsupervised WSD approaches have been more extensively studied and successfully applied in the context of our work on *concept-aware document classification* [TSW03, MTV+05] and semi-supervised learning, where the enhanced knowledge basis of a concept-aware classifier can substantially improve classification accuracy compared to a text classifier, in particular when only very little training data is available.

7.2.2 Similarity Joins

Edge Similarities

There have been various efforts proposed in the literature aiming to quantify semantic similarities of concepts in WordNet [Fel98]. We believe that among the most promising ones are those that aim to model concept similarities on

the basis of term and phrase correlations over large, real-world data collections. These measures exploit co-occurrence statistics for terms (or n-gram phrases) to estimate the semantic relatedness of terms and, hence, concepts in a given corpus. Ideally, this is the same corpus that is also used for querying. A measure often referred to for this purpose is the *Dice* coefficient.

As for Dice coefficients, the similarity between to senses S_1 and S_2 is defined as:

$$dice(S_1, S_2) \quad := \quad max_{S_1 \times S_2} \left\{ \; 2 \cdot \frac{df(t_{1,i} \wedge t_{2,j})}{df(t_{1,i}) + df(t_{2,j})} \right\} \tag{7.1}$$

where $t_{1,i} \in S_1$ and $t_{2,j} \in S_2$, respectively, and $df(t_{1,i} \wedge t_{2,j})$ is the cardinality of documents that contain both $t_{1,i}$ and $t_{2,j}$. Note that with the presence of an explicit thesaurus or dictionary, phrase indexing may become feasible, although it would remain a costly operation, because the indexer would have to implement the aforementioned window parsing technique for each potential token sequence in all documents of the collection. Moreover, subsequent phrase matching in query executions would be restrained to the ones contained in the particular thesaurus. Without explicit knowledge about the document frequencies of phrases, however, we may also use the maximum single-term co-occurrence frequency as an approximation to estimate the Dice similarities among concepts.

Note that Dice weights are precomputed for all related concept pairs (e.g., all hypernym/hyponym edges) in the concept graph. In most IR applications, these concept mappings will not be at hand for all terms and documents in the corpus. Therefore, we propagate the precomputed concept weights back to the term level after the WSD step. This way, retrieval is still performed on the term level, but term similarities are not static but rather depend of the disambiguation of the terms (or phrases) in the given document or query context (see Section 7.2.1 for details of the disambiguation step).

Since WordNet offers directed edges or relationships between concepts, another option for modeling these concept similarities would be conditional probabilities of the form $P[S_1|S_2]$. Yet the current implementation and experiments use precomputed Dice coefficients for their better way to generalize. We believe they are the more viable approach that reflects corpus-based correlations (which are undirected) in our intuitive feeling of semantic similarities in a better way. Moreover, Dice coefficients produce higher absolute values than conditional probabilities. Optionally, these may be smoothed with a similar method as described in Section 3.3.

Path Similarities

To implement the similarity search for two arbitrary, not directly connected concepts S_1 and S_2, we employ Dijkstra's shortest path algorithm [CLRC01] to find the shortest connection between S_1 and S_2. Then, interpreting the edge similarities as transition probabilities, the senses' final path similarity $sim(S_1, S_2)$ for a path $\langle v_1, \ldots, v_k \rangle$ of length k with $v_0 = S_1$ and $v_k = S_2$ and $\langle v_i, v_{i+1} \rangle \in V$ for $i = i, \ldots, k-1$ is defined as

$$sim(S_1, S_2) := \prod_{i=1}^{k-1} dice(v_i, v_{i+1}) \tag{7.2}$$

If there is more than one path that minimizes the length, we choose the one with highest path similarity sim to yield the final concept similarity. Note that regarding hypernym/hyponym relationships, WordNet's structure is primarily a small forest of large concept trees. For example, the common hypernym of all concrete noun concepts is "entity"; and there are only 8 more root concepts. Hence, we can take advantage of this observation and generalize the above approach by introducing an artificial super-root concept "all" that connects all the 9 original WordNet root concepts.

7.3 Unified Ontology Service

In [TSW03], we also defined a framework for a Unified Ontology Service that provides unified access to multiple thesaurus or ontology sources such as WordNet or OpenCyc [OPE] in a compact API. Our recent work also investigated the automatic ontology extraction from large, semistructured and richly linked Web collections such as the Wikipedia [WIK] encyclopedia with more than 800,000 documents with promising results. The developed service can be centrally deployed and accessed via a set of WebServices using the SOAP protocol for Remote Method Calls, as Enterprise Java Beans (EJB) using the Remote Method Invocation (RMI) of the Java environment, or locally via direct API calls.

The internal structure of our ontology service is derived from the WordNet graph and stored in a set of database relations for concepts, i.e., the nodes of the ontology graph that yield all known synsets S, and relations R_t for the 16 currently supported directed edge such as types hypernym, hyponym, holonym, meronym, antonym, pertainym, etc., that define the way how the synsets nodes are connected. For each edge type t, all edges of the ontology graph are stored as triples $R_t = \{(sid, tid, dice) \mid with\ sid, tid \in S;$ $dice \in [0, 1]\}$ for $t \in$ *{hypernym, hyponym, holonym, meronym, antonym,*

pertainym,...} including the identifiers of the source and the target synset and their estimated semantic similarity using Dice coefficients. The API facilitates the detection of n-grams and their disambiguation for a given set of context words, as well as the statistical quantification of concept similarities on the basis of a large (topic-specific) Web crawl or a benchmark-specific document collection.

The Ontology Service provides the following basic functions:

- `findConceptByTerm` – returns all matching concepts for a given string expression; this expression may be a single keyword or a phrase
- `disambiguateConcepts` – disambiguates a set of concepts (usually obtained from an initial `findConceptByTerm` query) for a given context string (e.g., part of a document or topic description)
- `findConceptSimilarity` – queries the concept graph to detect the shortest path between two concepts S_1 and S_2 and returns the aggregated edge similarity $sim(S_1, S_2)$
- `getDiceCoeff` – computes the Dice similarity for two arbitrary concepts S_1, S_2 referred to by their ids

7.4 Incremental Merge Operator

Given a set of p inverted index lists for an expansion set $exp(t_i) = \{t_{i1}, \ldots, t_{ip}\}$ that are already sorted in descending order of local scores, we can efficiently merge the p lists by iteratively moving a cursor through each of the input lists to get a sorted output list with minimal overhead of the merging algorithm itself. Then the $high_i$ score at the end of the incrementally created output list is an upper bound for all the local high values at the current cursors of the input lists.

If we also want to incorporate term similarities $sim(t_i, t_{ij})$ into the resulting output scores, we simply multiply the similarity values with the scores of the input lists during the merging procedure and still retain a sorted output list, since these similarities are constants at query runtime. If we further want to implement a *max-score aggregation* (as denoted below) that only allows the best match of a document for the whole expansion set to contribute in the score aggregation, we merely need to eliminate duplicate, i.e., repeated, document occurrences in the resulting output lists on-the-fly during the merging step. Since for each candidate d, we have to remember the set of evaluated dimensions $E(d)$ for the baseline top-k algorithm anyway,

we accept only the first occurrence of each document in the merged list for the worst- and bestscore updates and mark this information in $E(d)$. For the score aggregation, we skip all further occurrences of the same document in the merged output list, inevitably having a lower score.

By embedding this method into the TA-style baseline algorithm, we iteratively pull the next tuple from this virtual list without having to predetermine the final scan depth in the merged list. Through exploiting the given index structures for random access on plain lists in a smart way, we even provide efficient support for random accesses on the merged lists as required by our cost-based index access scheduler. Then there is no difference for the query processing strategy at the encapsulating top-k algorithm, regardless of whether it operators on a number of physically stored inverted lists, multiple virtual lists, or even a mixture of both kinds. Thus, the Incremental Merge algorithm can be implemented as new *join operator* that is natively embedded into the given infrastructure of top-k algorithms. Even all our previous assumptions about probabilistic candidate pruning and cost-based index access scheduling remain valid – with very few modifications, however, using a notion of *meta histograms* that capture the score distribution and selectivity of the merged lists (see Section 7.6).

7.4.1 Max-Score Aggregation

The top-k algorithm is extended such that it now merges multiple index lists L_{ij} in descending order of the combined score that results from the local score $s_{ij}(d)$ of an expansion term t_{ij} in a document d and the thesaurus-based similarity $sim(t_i, t_{ij})$. Moreover, to reduce the danger of topic drift, we use the following modified score aggregation for a query $t_1 \ldots t_m$ that counts only the *best match* for any one of the expansion terms per document:

$$score(d) := \sum_{i=1..m} \max_{t_{ij} \in exp(t_i)} \{sim(t_i, t_{ij}) \cdot s_{ij}(d)\}, \tag{7.3}$$

with analogous formulations for the $worstscore(d)$ and $bestscore(d)$ bounds as used in the baseline top-k algorithm.

The actual set of expansions is typically chosen such that for a query with terms $t_1 \ldots t_m$, we first look up the potential expansion terms $t_{ij} \in exp(t_i)$ for each t_{ij} with $sim(t_i, t_{ij}) > \theta$, where θ is a fine-tuning threshold for limiting $exp(t_i)$. It is important to note that this is not the usual kind of threshold used in query expansion. In our method, we can choose fairly small values θ and the exact choice is not critical. In contrast to a static expansion, our algorithm does not necessarily exhaust the full set and remains fairly robust for a broad range of θ, such that θ could even be set to 0 in the incremental

expansion setup. However, for the experiments we also need θ to initialize a set of static expansions that serves as a competing baseline reference.

7.4.2 Incremental Merge Algorithm

Algorithm 7 shows pseudo code for the Incremental Merge algorithm that is seamlessly integrated into the multi-threaded architecture of the TopX core query processor (see Section 4.3) and replaces the former `processIndexList` procedure of the index lists scan thread for physically stored inverted lists.

Then the index lists for the expansion terms for a given query term are merged on demand (and, hence, incrementally) until the *min-k* threshold termination at the enclosing top-k operator is reached, by using the following scheduling procedure for the index scan steps: the next scan step is always performed on the list L_{ij} with the currently highest value of $sim(t_i, t_{ij}) \cdot high_{ij}$, where $high_{ij}$ is the last score seen in the index scan (i.e., the upper bound for the unvisited part of the list). This procedure guarantees that index entries are consumed in exactly the right order of descending $sim(t_i, t_{ij}) \cdot s_{ij}(d)$ products.

This way, we are in a position to open the scans on the expansion-term index lists *as late as possible*, namely, when we actually want to fetch the first index entry from such a list. Thus, resources associated with index-scan cursors are also allocated on demand.

Index List Metadata

The pairwise similarities $sim(t_i, t_{ij})$ and the initial index list high-scores $high_{ij}$ constitute the index list metadata that is required to efficiently initialize the Incremental Merge algorithm.

The pairwise similarities $sim(t_i, t_{ij})$ for $t_{ij} \in exp(t_i)$ are retrieved by the Ontology Service upon query initialization. Note that the amount of data retrieved thereby is fairly small such that large parts of the thesaurus can actually be cached in main memory (e.g., all WordNet nouns and their path similarities). The $sim(t_i, t_i)$ for identical source and target expansions is defined to yield the maximum similarity value of 1, such that the Incremental Merge scans are initialized on the index lists for the original query conditions t_i first.

The maximum possible scores of all expansion candidates, on the other hand, i.e., the initial $high_{ij}$ values for all expansion terms $t_{ij} \in exp(t_i)$ at the start of the query execution, can be derived from the index lists histograms that are needed for the probabilistic candidate pruning or cost-based scheduling decisions, anyway. Note that even in the absence of this metadata, we

Algorithm 7 Incremental Merge Algorithm.

```
1: processIndexListIncremental(Expansion exp(t_i)={t_i1, ..., t_ip}, Similarities sim(t_i,t_i1), ...,
   sim(t_i,t_ip), Index list max-scores high_i1, ..., high_ip, Batch-size b_i)
2: // Initialize set of active expansions activeExp(t_i) for term t_i
3: activeExp(t_i) := {t_i};
4: best_i := t_i;
5: next_i := t_ij with max{sim(t_i,t_ij) | t_ij ∈ {exp(t_i) - t_i} };
6: nextHigh_i := sim(t_i, next_i) · high_nextExp_i;
7: // Determine the currently best expansion best_i and the threshold nextHigh_i
8: // for jumping to the next best expansion next_i
9: best_i := t_ij ∈ activeExp(t_i) with max{sim(t_i,t_ij) · high_ij};
10: if (sim(t_i, best) · high_best < nextHigh(t_i)) then
11:    activeExp(t_i) := activeExp(t_i) ∪ {best};
12:    best_i := next_i;
13:    next_i := t_ij with max{sim(t_i,t_ij) · high_ij | t_ij ∈ {exp(t_i) - best_i} };
14:    nextHigh_i := sim(t_i,next_i) · high_next_i;
15: end if
16: // Perform next sorted access to L_best_i
17: <docid, score> := L_best_i.getNext();
18: d := cache.getCachedItem(docid);
19: s_i(d) := sim(t_i, best_i) · score;
20: E(d) := E(d) ∪ {i};
21: high_best_i := s_i(d)
22: pos_i++;
23: // Continue as in scan thread of the core query processor in Alg. 4
24: ...
25: // Suspend & wait for main thread notification
26: if pos_i mod b_i == 0 then
27:    isSuspended_i = true;
28:    semaphore.notify();
29:    this.waitForNotification();
30: end if
31: isSuspended_i = false;
32: ...
```

are able to perform the Incremental Merge procedure; we simply have to fetch the first tuple of each index lists entry to get the initial $high_{ij}$ values, assuming the perfect default similarity $sim(t_i, t_{ij})$ of 1, however.

7.4.3 Sorted Access for Dynamic Expansions

The Incremental Merge algorithm naturally provides sorted access to the "virtual" index list that results from merging a set of physically stored or other virtual index lists in descending order of their combined $sim(t_i, t_{ij}) \cdot s_{ij}(d)$ scores. Figure 7.5 illustrates this merging procedure for an expansion $exp(t) = \{t_1, t_2, t_3\}$. The best initial combined score based on the index list metadata is determined to be 0.9 for term t_1. Then list L_1 is scanned until the combined score for that list falls below the next best combined score for another list, namely L_2 with 0.72 in this case. Then the algorithm remembers the combined score at the last scan position of L_1 and switches to the next

best expansion list L_2, and so on, until the threshold stopping condition at the top-level operator is reached.

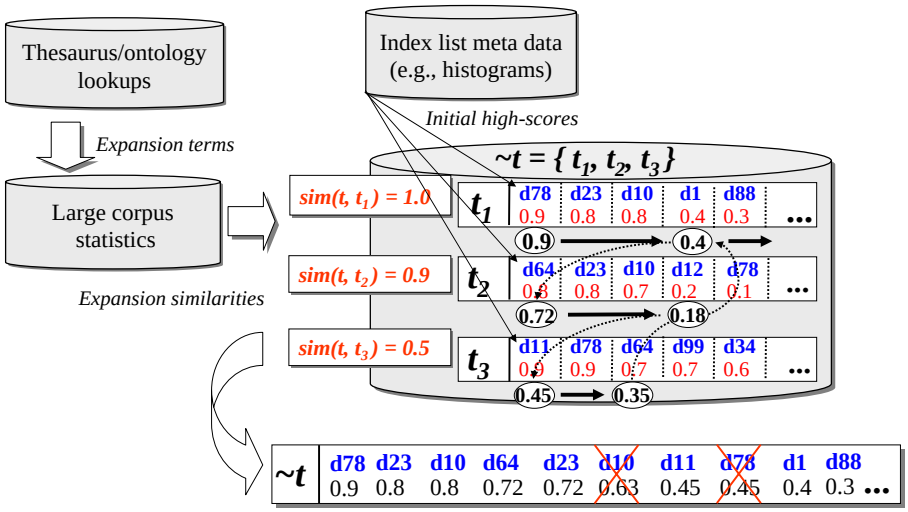


Figure 7.5: Example schedule for Incremental Merge.

Note that multiple occurrences (duplicates) of the same document in the merged list that arise from different expansion lists are eliminated on-the-fly as indicated by the red bars in Figure 7.5. This information is stored in the $E(d)$ sets for each candidate individually.

Index lists for potential expansion terms are opened and added to the set of active expansions $activeExp(t_i)$ for term t_i, only if they are beneficial for identifying a top-k candidate. Often, the scan depth on these lists is much smaller than on the index lists for the original query terms, and for many terms in $exp(t_i)$ the index scans do not have to be opened at all, i.e., typically the set of active expansions $activeExp(t_i) \subset exp(t_i)$ is a true subset of the whole expansion set.

7.4.4 Random Access for Dynamic Expansions

Random access support to a set of dynamically merged inverted lists requires specific consideration for the Incremental Merge operator, since we need to combine the precomputed local index scores with the query-specific expansion similarities for a potentially large expansion set *efficiently*, and we certainly cannot afford to materialize the whole merged expansion list at any time of the query processing. Fortunately, it turns out that with the B$^+$-tree index on the `TextFeaturesRA` base table over the key tuple $(docid, term)$ (see Section 2.3), we already made a safe choice, because the range of the random

access scan still is primarily bounded by the document id. Assuming $N \gg M$, using this attribute combination remains much more efficient than an index on the key $(term, docid)$. Figure 7.6 shows the respective SQL statement for a set of active expansions containing the terms "car", "truck", etc.

```
select
  term, score
from
  FeaturesRA
where
  docid = '12345' and term in ('car', 'truck', ...)
```

Figure 7.6: Index range scan on the `TextFeaturesRA` text index as random access statement for a given document and expansion list.

This way, random accesses remain fairly efficient and are explicitly supported for the Incremental Merge method as well, because the query processor just has to fetch all term-score pairs by a primary index range scan on the *docid* attribute and some secondary, intermediate index skip scans on the *term* attribute for each expansion term. The term-score pairs can be combined with the concept similarities in-memory by the query processor in order to support the max-score aggregation. Note that the above statement returns tuples only for those expansion terms that are actually present in the current document which is typically a small subset of the expansion. Again, the structure of these statements can be precompiled and uploaded to the DBMS prior to query executions for the whole expansion set.

7.5 Nested Top-k Operator

For more sophisticated query subconditions such as multiple *phrase expansions*, local scores for individual conditions cannot be fetched from materialized index lists but need themselves to be computed dynamically. This poses a major problem to any top-k algorithm that wants to primarily use sorted accesses. A possible remedy would be that the global top-k operator "guesses" a value k' and asks the dynamic source to compute its top-k' results upfront, with k' being sufficiently large so that the global operator never needs any scores of items that are not in the local top-k'.

Note that, for example, the rank-aggregation framework [IAE03, ISA+04, LCIS05, LSCI05] pursues this strategy for splitting an m-dimensional query into a tree of binary, conjunctive joins. In order to provide a safe upper bound

of k', a Uniform distribution of scores in the basic input lists is assumed which may lead to overly conservative upper bounds for k' and unnecessarily deep scans on the underlying input sources, whereas top-k-style query processing would just make us expect the largest benefits for non-Uniform, skewed input scores. Moreover, we do not restrict ourselves to binary, conjunctive top-k operators, but we consider index lists to be equally relevant to the query altogether. So the demand for nested top-k operators in our case rather arises from specific expansion tasks such as a mixture of phrase and single-keyword expansions that are merged incrementally, and with lists for individual phrases being combined dynamically and only on demand, thus translating the explicit structure defined by a query and its expansion sets into a highly specialized operator tree.

In general, we believe that the common fix of estimating a k' value for subordinate top-k operators is unsatisfactory, since it inherently is very difficult to choose an appropriate (i.e., safe and tight) value for k', and this approach destroys the incremental and pipelined nature of the TA-style methods.

7.5.1 Dynamic Index Lists

TopX treats such situations by running a nested top-k operator on the dynamic data source(s), which iteratively reports candidates to the caller (i.e., the global top-k operator), and efficiently synchronizes the candidate priority queues of caller and callee. The callee starts computing a top-∞ result in an incremental manner, by whatever means it has; in particular, it may use a TA-style method itself without a specified target k, hence top-∞. It gradually builds a candidate queue with $[worstscore'(d), bestscore'(d)]$ intervals for each candidate d. The caller periodically polls the nested top-k operator for its currently best intermediate results with their score intervals. Now the caller integrates this information into its own bookkeeping by adding bestscores to the bestscores of its global candidates and worstscores to the worstscores of its global candidates. From this point, the caller's processing simply follows the standard top-k algorithm (but with score intervals). When the global rank-k worstscore is at least as high as the highest bestscore among all non-top-k candidates at the global level, the caller terminates and notifies all nested top-k operators.

This method nicely provides a non-blocking pipelining between caller and callees, and gives the callees leeway as to how exactly they proceed for computing their top results. Note that the caller may terminate (and terminate all callees) long before a callee has really computed its final top results. Within the TopX engine, nested top-k operators are primarily useful for handling phrase matching.

7.5.2 Phrase Matching

In general, it will be too expensive to precompute and materialize an inverted list for all possible phrases, i.e., multi-keyword composite terms that a query wants to find as adjacent words. But if we merely index the individual words, we cannot simply read off the combined scores from local index lists. Furthermore, scanning the index lists for all relevant words indeed provides us with candidates, but we need an additional adjacency test to eliminate false positives. Moreover, if we consider adjacency as a relaxable condition (e.g., allowing the words of a phrase to appear within some short distance), there is an additional aspect in the scoring function that may violate the monotonicity of the score aggregation that top-k-style algorithms rely on. Our solution is to encapsulate phrase conditions in separate top-k operators and invoke these from the global top-k operator in the pipelined manner described above.

Recall from Section 4.2.3 that we store term positions in a separate index (not in the inverted index lists themselves) to keep the actual inverted index as compact as possible and minimize disk I/O when scanning these lists. Thus, testing the adjacency condition (or other kinds of proximity conditions and computing an additional proximity factor for the overall score contribution) requires random I/O. We therefore further treat the adjacency tests as expensive predicates in the sense of [CwH02] and postpone their evaluation as much as possible.

Now, for the Incremental Merge expansions with multiple phrases, we incrementally merge lists of partially evaluated candidates obtained from a separate nested top-k operator for each phrase in descending order of candidate bestscores. For single-keyword expansions, the score obtained from a single inverted list will already be the final score for that candidate and expansion; for phrase expansions, the score will be a partial score obtained from one or more local keyword conditions of the phrase. Just like in the non-expanded case, our approach assumes that the score for a phrase match of a phrase $t_1 \dots t_p$ is the sum of the scores for the individual words, possibly normalized by the phrase length, thus reducing the otherwise overly big impact of long phrases. The key idea to minimizing the random I/Os for adjacency tests is to eliminate many candidates based on their upper score bounds (or estimated quantile of their scores) early before checking for adjacency. Consider a phrase $t_1 \dots t_p$ and suppose we have seen the same document d in the index lists for a subset of the phrase's words, say in the lists for $t_1 \dots t_j$ ($j < p$). At this point we know the $bestscore'(d)$ with regard to this phrase match alone (which is part of a broader query), and we can compare this $bestscore'(d)$ or the total $bestscore(d)$ for the complete query

against the $worstscore(d')$ of other candidates d' that have been collected so far. In many cases, we can eliminate d from the candidate pool without ever performing the adjacency test. Even if we have seen d in all p index lists for the phrase's words, we may still consider postponing the adjacency test further, as we gain more information about the total $bestscore(d)$ for the entire query (not just the phrase) and the evolving *min-k* threshold of the current top-k documents. This seamless integration of multiple Incremental Merge and nested top-k operators into complex operator trees provides a very finely grained modeling of semantic similarities at the very core level of query processing.

Phrase Expansion Example

Figure 7.7 depicts an expansion for the keyword query "undersea fiber optics cable" which is another "hard" query from the TREC 2004 Robust track. The phrase "fiber optic cable" has been automatically identified as a WordNet concept and yields the very near match "fiber optics" with a similarity value of 0.8, assuming stemming is disabled for the inverted lists. So we can automatically rewrite this query, using the original keyword condition "undersea" and an Incremental Merge over the two phrases "fiber optic cable", i.e., the original condition with similarity 1, and the actual expansion "fibre optics", each using a nested top-k operator to dynamically combine the inverted lists with regard to each phrase condition.

7.5.3 Nested Top-k Algorithm

The algorithm works by running outer and inner top-k operators in separate threads, with inner operators periodically reporting their currently best candidates along with the corresponding $[worstscore_i(d),\ bestscore_i(d)]$ intervals. Then the aggregated score interval of a candidate d at each operator simply becomes the *sum* of the score interval boundaries propagated by each of the subordinate operators

$$\left[\sum_{i=1..m} worstscore_i(d),\ \sum_{i=1..m} bestscore_i(d)\right]$$

for a top-k join, or the *maximum* of the respective interval boundaries

$$\left[\max_{i=1..m} worstscore_i(d),\ \max_{i=1..m} bestscore_i(d)\right]$$

for an Incremental Merge join which remains a monotonous score aggregation at any level of the operator tree.

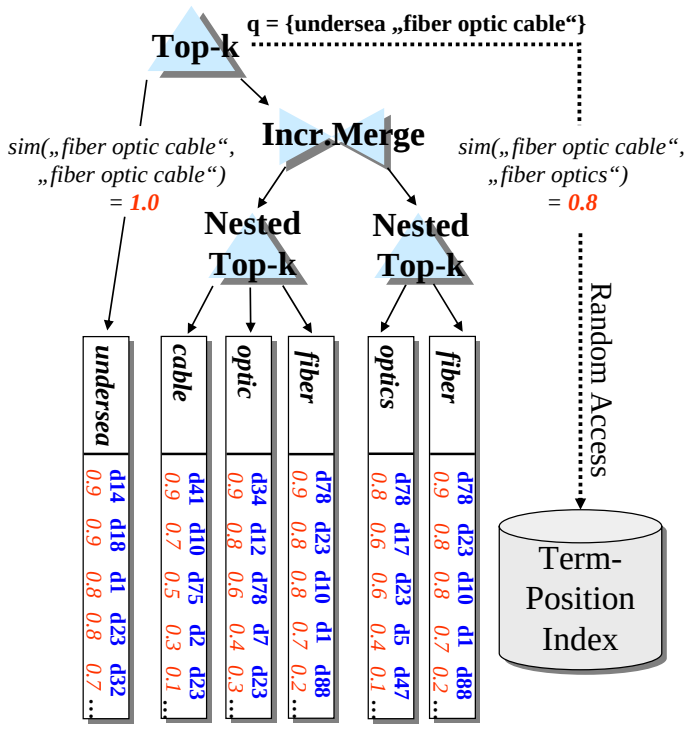


Figure 7.7: Phrase matching with multiple nested top-k operators.

Figure 8 shows pseudo code for the nested top-k algorithm. Just like the previous top-k operator, the nested top-k operator gathers candidates in a queue with candidates ordered in descending order of bestscores. Note that since the sum of the current $high_i$ values is monotonically decreasing, the bestscores of all yet unseen candidates are monotonically decreasing, too. This way, the lower operators queue serves as a buffer from which partly evaluated candidates can be read and propagated to the upper operator in the sense of a *sorted access*. The superordinate operator may in turn be a (nested) top-k operator or an Incremental Merge operator. For dynamic query expansion with phrases, this is always an Incremental Merge operator. Note that the nested top-k operator is initialized with $k = \infty$, it just periodically prefetches a batch of sorted accesses from the subordinate operators or physically stored index lists, typically using the same batch size b that is scheduled for the top-level operator. It then falls into the very same synchronization loop that is provided by the responsible superordinate scan thread and waits for notification by the superordinate operator's main thread (see Section 4.3). This way, the nested top-k operator's main thread becomes the scan thread of the parent top-k operator for the respective query dimension.

Algorithm 8 Nested Top-k Algorithm.

```
1:  processIndexListNested(IndexList L_i, Batch Size b_i, Operator parentTopk, Parent dimension
    parentDim)
2:  isAlive_i = true;
3:  isSuspended_i = false;
4:  pos_i = 0;
5:  while isAlive & L_i.hasNext() do
6:     // Perform next sorted access to L_i
7:     <docid, score> := L_i.getNext();
8:     d := cache.getCachedItem(docid);
9:     s_i(d) := score;
10:    E(d) := E(d) ∪ {i};
11:    high_i := score;
12:    pos_i++;
13:    // Update worst- and bestscore bounds
14:    worstscore(d) := Σ_{i∈E(d)} α_i(β_i+s_i(d));
15:    bestscore(d) := d.worstscore + Σ_{ν∈Ē(d)} α_ν(β_ν + high_ν);
16:    // Fetch d^parent from parentTopk operator and
17:    // propagate the score update directly to the parent operator
18:    d^parent := parentTopk.cache.getCachedItem(docid);
19:    if parentDim ∈ parentTopk.E(d^parent) then
20:       worstscore_parentDim(d^parent) := worstscore(d);
21:       bestscore_parentDim(d^parent) := bestscore(d);
22:    else
23:       candidates.insert(d);
24:    end if
25:    // Suspend & wait for main thread notification
26:    if pos_i mod b_i == 0 then
27:       isSuspended_i = true;
28:       semaphore.notify();
29:       this.waitForNotification();
30:    end if
31:    isSuspended_i = false;
32: end while
33: isAlive_i = false;
```

Random access may be supported as well, thus propagating the random lookup down the operator tree until to the leafs nodes of the scans which always refer to physically stored index lists that are accessed through some kind of B^+-tree index structure.

At the synchronization points, the outer operator integrates the reported information into its own candidate bookkeeping and considers pruning candidates from the outer priority queue. The top-level operator's queue is also the only point-of-attack, where probabilistic candidate pruning and index access scheduling is applied on the basis of index list statistics and global candidate scores.

Incremental Merge and Nested Top-k as Eddies

The incremental evaluation of candidates by a nested operator and the propagation of $[worstscore_i(d), bestscore_i(d)]$ intervals at each query dimension i requires the lower operator to remember whether a candidate has been read off its queue already, which is again achieved through checking the $E(d)$ set of each candidate at the parent top-k operator. If $i \notin E(d)$, d has not been taken into the parent top-k's score aggregation for d, and d is first enqueued at the nested operator; if otherwise $i \in E(d)$, d has already been read off the nested top-k operator's queue. Enqueuing d at the nested operator again would not be beneficial, since it might be considered as a duplicate by the parent operator in case of Incremental Merge, and the update would be lost. Rather, the nested top-k has an option to *directly propagate* the score update "upwards" the operator tree until to the root operator (see Algorithm 8).

The direct propagation of score updates to the parent top-k operator provides a means to break the operator pipeline and allows to update a candidate's score directly, i.e., whenever the lower operator at dimension i gains new information on d's $[worstscore_i(d), bestscore_i(d)]$ bounds which may lead to undelayed candidate pruning and at the same time keeps all score updates monotonous. Note that this notion of individual tuple routing is very related to the Eddies [AH00, DH04] architecture. In fact, we could encapsulate the Incremental Merge and nested top-k technique into an Eddy-style execution plan for each candidate individually, using the max-score aggregation as described above.

Lazy Phrase Validations

Our technique also allows a lazy scheduling of random lookups for phrase tests by applying the expensive predicate test at the top-level operator, only, which can further decrease the amount of phrase tests by an order of magnitude (as indicated by our experiments in Section 9.7.3) for large phrase expansions.

Mandatory and Optional Expansion Groups

The modified top-k algorithm with Incremental Merge generalizes the idea of boosting factors for individual terms to whole expansion groups using the modified score aggregation $\sum_i^m sim(t_i, t_{ij}) \cdot (\beta_i + s_{ij}(d))$. This may lead to increased scan depths on the corresponding index lists, if an otherwise promising result candidate has very low scores in these lists but needs to be tested for the presence of a mandatory term t_{ij} out of an expansion set $exp(t_i)$. The β_i coefficients are either 0 for the default "andish" interpretation of the expansion group or set to a high value, i.e., $\beta_i \geq 1$, for mandatory

terms whose local score should not effortlessly be compensated by other query conditions (see also Section 3.5 for details on incorporating query weights and term boosting for plain keyword queries).

7.6 Probabilistic Extensions

When we want to employ probabilistic pruning based on score predictors, we face an additional difficulty with query expansion using the Incremental Merge technique, since the original histograms or parameterized score estimators no longer capture the resulting score distribution for a set of incrementally merged lists.

Before computing the convolution over the still unknown scores for a subset of the original query's terms, we need to consider the possible expansions for each of the original query terms. For a term t_i with a random variable S_i capturing the score distribution in the not yet visited part of the incrementally merged lists, we are interested in probabilities of the form $P[S_i > x \mid S_i \leq high_i]$. With expansion into $exp(t_i) = \{t_{i1}, ..., t_{ip}\}$, we obtain

$$P[\ max\{sim_{i1} \cdot S_{i1}, \ldots, sim_{ip} \cdot S_{ip}\} \leq x \mid S_{i1} \leq high_{i1}\ \forall t_{ij} \in exp(t_i)\ ] \tag{7.4}$$

where sim_{ij} is shorthand for $sim(t_i, t_{ij})$. This is equivalent to

$$\prod_{j=1}^{p} P[\ sim_{ij} \cdot S_{ij} \leq x \mid S_{ij} \leq high_{ij}\ \forall t_{ij} \in exp(t_i)\ ]\ , \tag{7.5}$$

assuming independence between score distributions of different lists. Note that this independence assumption is unlikely to hold on real-life data, but it already seems to go a long way in terms of accuracy and resulting performance gains for small expansion sizes p. We thus obtain

$$1 - \prod_{j=1}^{p} P[\ sim_{ij} \cdot S_{ij} > x \mid S_{ij} \leq high_{ij}\ \forall t_{ij} \in exp(t_i)\ ] \tag{7.6}$$

As the individual factors of this product are captured by the precomputed histograms per index list (with the constant $sim(t_i, t_{ij})$ coefficients simply resulting in a proportionally adjusted distribution), we can now easily derive a new combined histogram for the max distribution.

7.6.1 Selectivity Estimator for Incremental Merge

Unfortunately, for increasingly large expansions, the inherent independence assumption in the predictor derived above would dominate the score estimations which would converge to 1 regardless of the actual score distributions in the basic input lists. This would render the candidate pruning without effect.

Moreover, postulating a basic parameterized score distribution for the merged lists out of the distribution parameters of the input lists is not easy to justify anymore. So, for the following subsections, we will focus on the extraction of *meta histograms* with the capability to capture arbitrary distributions for the max-score aggregation over multiple merged lists. In order to get a clue on the resulting distribution that the Incremental Merge technique creates, let us first have a look at what goes on when the Incremental Merge would not be embedded into a top-k algorithm but would completely merge all input lists.

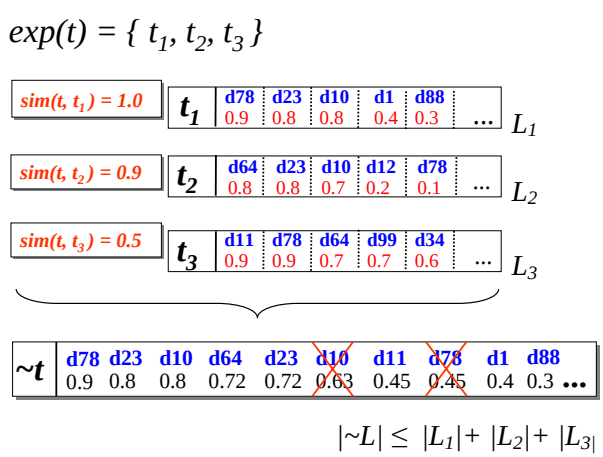


Figure 7.8: Estimated selectivity of a set of merged lists.

Figure 7.8 depicts the situation for our example Incremental Merge schedule from the previous section. Ignoring overlap, the full length of the resulting merged list for an expansions $exp(t_i) = \{t_{i1}, \ldots, t_{ip}\}$ is upper bounded by the sum of the lengths of the input lists:

$$\widetilde{|L_i|} \leq \sum_{t_{ij} \in exp(t_i)} |L_{ij}| \ . \tag{7.7}$$

This may already be a good-enough guess for top-k scheduling and probabilistic pruning applications, although it tends to substantially overestimate the selectivity of the merged lists, since our algorithm eliminates du-

plicate document occurrences with respect to the max-score aggregation. Without any further information about feature correlations or co-occurrence statistics, we may in fact choose to rely to this simple estimate.

Incremental Merge Selectivity with Feature Correlations

However, with the presence of pairwise correlation statistics, or more specifically, with term-co-occurrence values for $|L_i \cap L_j|$ as described in Section 6.2.3, we can significantly improve the first estimate through a correction for the pairwise overlap as

$$\widetilde{|L_i|} \approx \sum_{j=1}^{p} |L_{ij}| + |L_{i\,j+1}| - |L_{ij} \cap L_{i\,j+1}| \tag{7.8}$$

$$\leq \sum_{t_{ij} \in exp(t_i)} |L_{ij}| \tag{7.9}$$

7.6.2 Meta Histograms for Incremental Merge

As for estimating the score distribution that the Incremental Merge algorithm creates for an expansion $exp(t_i) = \{t_{i1}, \ldots, t_{ip}\}$, we employ the basic input histograms, thus covering the score range $(0, 1]$ with n equi-distant buckets $(lb_k, ub_k]$ with $lb_k = k/n$ and $ub_k = (k+1)/n$. Recall from Section 5.2.3 that each cell k stores the frequency $freq[k]$ and the cumulative frequency $cfreq[k]$ of scores that fall into the interval $(lb_k, ub_k]$. So, given the input bucket frequencies

$$freq_{ij}[k] = \frac{\#docs\ d \in L_{ij}}{|L_{ij}|} \quad with\ s_{ij}(d) \in \left(\frac{k}{n}, \frac{k+1}{n}\right] , \tag{7.10}$$

and expansion similarities $sim(t_i, t_{ij})$, we can construct the meta histogram's bucket frequencies as a weighted sum of the input frequencies

$$\widetilde{freq_i}[k] = \sum_{t_{ij} \in exp(t_i)} \left(\frac{|L_{ij}|}{\sum_{\nu=1}^{p} |L_{i\nu}|} \cdot \sum_{l=\lceil \frac{lb_k}{sim_{ij}} \rceil \cdot n}^{\lceil \frac{ub_k}{sim_{ij}} \rceil \cdot n} freq_{ij}[l] \right) \tag{7.11}$$

and

$$\widetilde{cfreq_i}[k] = \sum_{l=0}^{k} \widetilde{freq_i}[l] \tag{7.12}$$

which approximately captures the score distribution of the merged list, assuming overlap to be uniformly distributed among all scores and histogram

buckets. Here,

$$\sum_{l=\lceil \frac{lb_k}{sim_{ij}} \rceil \cdot n}^{\lceil \frac{ub_k}{sim_{ij}} \rceil \cdot n} freq_{ij}[l]$$

creates a proportionally adjusted distribution for each of the basic input distributions with input frequencies $freq_{ij}[0..n–1]$, thus taking the individual expansion similarities $0 < sim_{ij} \leq 1$ into account. Note that this estimate already forms a perfect probability distribution, with the sum of all bucket frequencies adding up to 1.

Meta Histograms with Feature Correlations

For considering feature correlations in the meta histogram construction, we would simply use the $\widetilde{|L_i|}$ estimates to derive the absolute frequencies of documents that are below (or beyond) a certain score as $\widetilde{|L_i|} \cdot \widetilde{cfreq_i}[k]$ (or $\widetilde{|L_i|} \cdot (1 - \widetilde{cfreq_i}[k])$, respectively).

Note that binary feature correlations already provide the major step from the first trivial estimate on our way to more precise selectivity estimations for the incrementally merged lists. Assuming overlap to be non-uniformly distributed among histogram buckets (and thus to depend on the scores) would require us to maintain more specific histograms for the pairwise list overlaps in addition to the basic input histograms. A perfectly precise selectivity estimator for the merged lists, however, would require all basic histogram buckets to contain the actual object identifiers of items that have been cumulated into each bucket's frequency value – which would not be in the sense of a *compact* histogram representation anymore.

Dynamic Meta Histogram Construction

This *meta histogram* construction is performed prior to query execution, and it can efficiently be updated whenever a new scan on another index list is opened with linear costs $O(n\,m')$ in the number of histogram cells n and the current number of active query expansions m'. Then this meta histogram is fed as input into the convolution with other (meta) histograms for the original query terms (or their expansion sets). In our experiments, the overhead for this dynamic histogram construction was negligible compared to the costs saved in physical disk accesses.

Summing up, we identify two orthogonal dimensions in which we may exploit feature correlations: one is query driven, namely the adjusted selectivity

estimation of *top-level* query conditions at a candidate's remainder dimensions; and the other is expansion driven, namely the adjusted selectivity estimation for an *incrementally merged* set of active expansion terms. Both are seamlessly integrated by the meta histogram approach which generalizes our previous assumptions for optimizing static queries.

Chapter 8

Top-k Query Processing for XML

Non-schematic XML data that comes from many different sources and inevitably exhibits heterogeneous structures and annotations (i.e., XML tags) cannot be adequately searched using database query languages like XPath or XQuery. Often, such queries either return too many or too few results. Rather the ranked-retrieval paradigm is called for, with relaxable search conditions and quantitative relevance scoring. Note that the need for ranking XML queries goes beyond adding Boolean text-search predicates to XQuery. In fact, similarity scoring and ranking are orthogonal to data types and would be desirable and beneficial also on structured attributes such as time (e.g., approximately in the year 1790), geographic coordinates (e.g., near Paris), and other numerical and categorical data types (e.g., numerical sensor readings and music style categories).

Research on applying IR techniques to XML data has started five years ago [CK01, FG01, SM02, TW00] and has meanwhile gained considerable attention. An emphasis of this thesis is to efficiently support vague search on element names and terms in element contents in combination with XPath-style path conditions. Structural similarity is considered in the sense that documents can qualify even if they do not satisfy all path conditions (if there were too few results otherwise). For relaxing tag names and content terms, thesaurus-based similarity measures are employed, and queries can be appropriately expanded. Boolean XPath-like query evaluations over content-and-structure are supported as well, but they inherently incur high evaluation cost for a top-k query processor that largely benefits from "andish" score aggregations and on-the-fly compensation of weak or non-existent matches to some query conditions.

8.1 Challenges in Efficient XML IR

Generalizing the algorithmic paradigm of top-k threshold algorithms from simple inverted lists to ranked XML retrieval is all but straightforward. The difficulties in applying the existing approaches to XML data lie in

1) the need to consider scores for XML elements while aggregating them at the document level,

2) the combination of vague content conditions with XPath-like structural conditions and efficient structural joins, thus dealing with uncertainty about structural and content related query conditions,

3) the adaptation of selectivity estimations and index access scheduling for both content and structure conditions and their impact on the evaluation strategies, and

4) the need to relax query conditions and the dynamic expansion of content-related and structural query conditions with similar constraints, if too few results satisfy all query conditions.

TopX addresses these issues by precomputing score and path information in an appropriately designed index structure, by largely avoiding or postponing the evaluation of expensive path conditions so as to preserve the sequential access pattern on index lists, and by selectively scheduling random accesses when they are cost-beneficial. As in the text case, TopX can compute approximate top-k results for XML using probabilistic score estimators, thus significantly speeding up queries with a small and controllable loss in retrieval precision. We leverage the methods presented in the previous chapters such as probabilistic candidate pruning, index access scheduling, and dynamic expansion techniques for a novel and versatile view on efficient XML retrieval.

8.1.1 An XML IR Example Scenario

Table 8.1 depicts an example query written in the NEXI syntax (see Section 2.2), with its corresponding tree structure shown in Figure 8.2. The gray-marked paragraph node in the figure denotes the *target node* of the query tree. All the remaining inner nodes are referred to as *support elements*. According to the query, we are looking for paragraphs (`par`) which are nested into some `article` element. The left branch of the query requires the *article* to have a bibliographic reference containing the term "W3C" as text content under the path `//article//bib//item` and an `//article//sec//par`

path pointing to a text node that is about "native XML databases". The `par` target is supposed to be embedded into a `sec` element that mentions the terms "XML retrieval" somewhere within that particular section element but not necessarily in the same paragraph as the actual target paragraph. Looking at this query as a tree, we see that its structure can be broken down into two basic twig structures, i.e., we have two branching elements, namely the `article` and the `sec` elements. Unlike in text retrieval, any subsequent matching of the content-conditions for XML scoring has to take these structural constraints into account, thus making XML top-k queries much more expensive than simple text queries.

```
//article[
  //bib[
    about(.//item, "W3C")
  ]
]//sec[
  about(.//, "XML retrieval")
]//par[
  about(.//,"native XML databases")
]
```

Figure 8.1: A NEXI query.

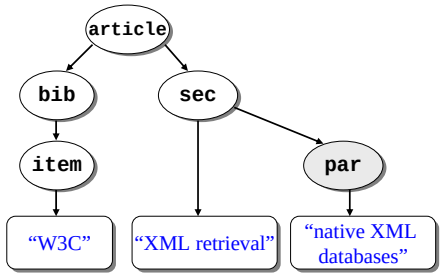


Figure 8.2: And its tree structure.

Among the two example documents shown in Figure 8.3, document d_1 is nearly a perfect match to the query, because the query pattern can be embedded into the document's structure almost as a whole. The only flaw of d_1 is that the matching element for the content condition about "XML retrieval" is not contained in the *same* `sec` element as the actual target paragraph for the content condition about "native XML databases", but it is indeed contained in a sibling section. This minor twist in the structure should be reflected by a slight penalty in the aggregated score for the document's target element.

The second document d_2, however, completely lacks any matching element for the "native XML databases" content conditions which is even specified to belong to the target element of the query and, therefore, should get a much lower score than d_1 – in fact even 0 according to our scoring model. Note that a strictly conjunctive, Boolean XPath-like evaluation of the query would have returned none of the two documents as a valid match for this query.

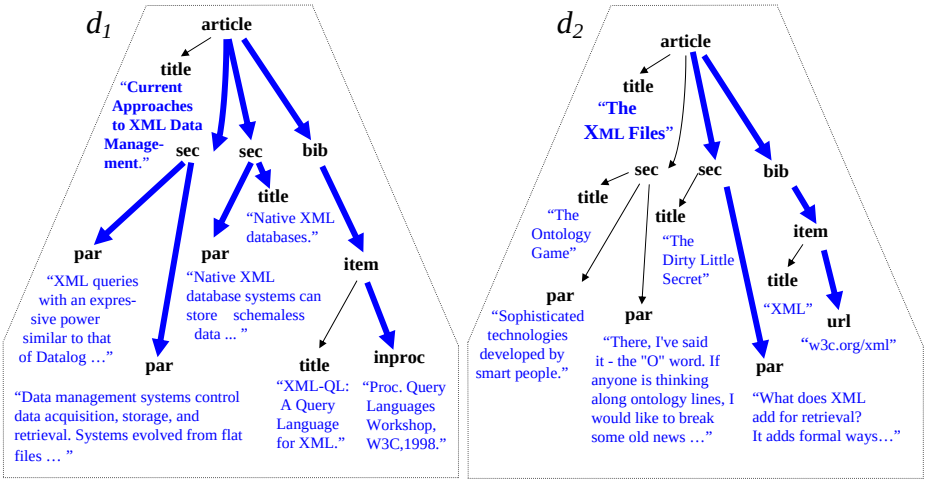


Figure 8.3: Two XML example documents of different relevance for the query of Figure 8.1.

8.1.2 Requirements & Solutions Overview

We briefly summarize the requirements for efficient XML IR together with our key solutions for the implementation of a high-performance, scalable top-k query processing engine for XML data as follows:

1) *Indexing:* Path conditions that test partial results for their connectivity along the specified XPath axes must be inexpensive to evaluate.

   – *Low Target Element Selectivity:* Full-content indexing of tag-term pairs accelerates the descendant axis for lowly selective target elements. We immediately benefit from less selective combined tag-term pairs and shorter inverted lists.

   – *High Support Element Selectivity:* Navigational tags in support elements aggregate additional score mass for the target element results and can be evaluated in the notion of expensive predicates. Highly selective target elements (as in INEX) do not decelerate query evaluations.

2) *Query Rewriting and Structure-Aware Processing:* We would like to achieve fast convergence of worst- and bestscores of candidate items even when we do not have the full knowledge about the candidates' complete structure for effective pruning and fast termination of index scans.

- *Query Dimensionality Reduction:* Merging of terms with their preceding tags helps reducing the query dimensionality without changing its semantics. Combining tag-term pairs benefits from lower combined selectivities and shorter inverted lists.
- *Deterministic and Monotonous Score Updates:* In order to provide a correct algorithm, the scoring model may not interfere with any structure- or scheduling-specific join-ordering, i.e., the order in which newly discovered elements are joined to the partial paths.
- *Early Worst- and Bestscore Guarantees:* We aim to provide tight and accurate, structure-aware score bounds for partially evaluated candidates at an early stage of the sequential scans.

3) *Random Access Scheduling:* The random accesses to test structural query conditions are expensive and should be minimized.

   - *Structural Selectivity Estimator:* According to a our cost model, we aggressively schedule lookups to structural query conditions in ascending order of selectivities at the end of the sequential scans.
   - *Structure-aware Candidate Pruning:* Lowly selective structural conditions can be assigned a high score. If any condition cannot be satisfied by a candidate, this candidate is most probably eligible for pruning and not evaluated any further.

4) *Dynamic Relaxation and Expansion of Query Conditions:* We primarily pursue an IR-style "andish" ranked retrieval for XML data. Content and structure conditions can be dynamically relaxed (or compensated) by a good match at another query condition and/or expanded in the sense of a similarity ($\sim$) operator, thus following the specification of the `ftcontains` operator of the XPath 2.0 Full-Extension and the (implicit) definition of the `about` operator in NEXI. *Efficient* query expansion for XML data poses further challenges to the query processing.

   - *Incremental Merge for Semistructured Data:* We generalize the Incremental Merge technique for dynamic and self-tuning query expansion to the XML case.
   - *Dynamic Selection of Index Access Structure and Hybrid Indexes:* Using tag-term pairs and tag sequences with pre/postorder labels to encode entire paths may not always be beneficial. We adapt the Incremental Merge technique to also incorporate hybrid index structures, including DataGuides [GW97], to also efficiently support lowly selective support elements.

XML Indexing

The first requirement is met by adding pre- and postorder values to the entries in the base tables (see Section 2.3), thus following the XPath accelerator work by [Gru02, GvKT03] which perfectly suits our requirements for compact path encodings that can be stored directly in the inverted lists. This gives us an efficient in-memory test as to whether an element e_1 is an ancestor of another element e_2 (within the same document) by evaluating $pre(e_1) < pre(e_2)$ and $post(e_1) > post(e_2)$, with analogous support for all XPath axes, including the child axis by extending this schema with the *level* information. Since the pre/post information is at hand as we continuously fetch the entries from the inverted lists for the content-related query conditions, we can test path conditions for candidates with high local scores without additional random lookups on a separate index for the structure. If an element fails such a path test, we simply drop it and exclude it from further structural joins. As this element itself may have been chained to other elements at different (yet unevaluated) query dimensions, dropping that element may have a crucial impact to the candidate's aggregated bestscore and, thus, prune a whole group of target elements for that document from the candidate queue.

Structure-aware Top-k Query Processing

As for the second requirement, fast convergence of worstscores and bestscores for candidate elements or entire documents, can be met by *block-scanning* all elements per query condition and, thus, eagerly eliminating uncertainty about further matches among documents for which only a subset of elements has been seen so far in the sequential index scans. The problem is that we may have to keep the document in the candidate queue for a long time until we finally find low-scored elements that might satisfy the content conditions but violate the path conditions. In particular, the bestscores would have to be kept unnecessarily high, namely at the aggregation of the maximum local term scores of elements at each dimension that are still a potential match for the path query and therefore are still valid. Analogously, the worstscores could still gradually increase as we find better elements that, though having a low local score themselves, could provide a crucial path connection and thus yield a higher aggregated score with regard to the whole path. As opposed to the text retrieval case, the worst- and bestscore bounds in the XML case would remain much too broad – and in fact almost meaningless – for candidate pruning. In using sequential block-scans, we make sure we have seen all the elements for a particular query condition as soon as we have encountered a document in one of the inverted lists; and this allows us to

reason about the worst- and bestscore bounds more accurately.

Random Access Scheduling for Structural Conditions

As for the third issue, TopX postpones random accesses as much as possible, namely, until the point when random I/Os are cost-beneficial according to our scheduling approach (see Section 8.4) for a given candidate. For most of the candidates, the content-based bestscore quantiles (or the probabilistically estimated quantiles) already become low enough, so they can be dropped from the candidate queue before the structural conditions need to be evaluated at all. This results in major savings of random accesses. We extend the beneficial cost model introduced in Section 6.4.2 for plain inverted lists for structured data and queries by analyzing and precomputing the structural selectivities of basic query patterns such as path and twig relationships of elements in the corpus.

Dynamic Relaxation and Expansion of Query Conditions

Finally, the fourth requirement is addressed by adopting the Incremental Merge approach for semistructured data. Applying the Incremental Merge and nested top-k framework for XML is fairly straightforward; the only principle obstacle is that query conditions are now constrained by structural path conditions. That is, the first maximum scored single element or whole element block per candidate that is reported by the Incremental Merge operator does not necessarily maximize the aggregated path score (and hence the document score) any more, similarly to the discussion above. Lower-scored elements that are only merged further down the lists, on the other hand, might contribute to a higher aggregated path score. We have to ensure that we incrementally allow further element blocks to be merged into the existing element block of a candidate for the structural joins and still keep the worst- and bestscore updates monotonous. As an additional tweak, we may exploit the Incremental Merge framework not only for *content-related query expansions* but also to *dynamically relax the structural query constraints* up to the usage of hybrid index structures, where some navigational query conditions correspond to pre/postorder labeled tag conditions and others contain DataGuide-like path encodings.

8.1.3 Boolean XPath vs. XML IR

Conjunctive Query Evaluation – Boolean XPath

For conjunctive or Boolean XPath-like top-k query processing, reasoning about the scores of partial matches with uncertain knowledge about the whole structure is a secondary issue. To evaluate, score, and rank a candidate item in conjunctive mode, we could just require the candidate to be fully evaluated at all query dimensions through a mix of sequential and random index accesses and then process the query on the whole document structure. This is as it is done in existing XPath and top-k query processors for XML [KKNR04]. Hence, in contrast to a traditional DBMS whose query optimizer is eagerly driven by selectivity assumptions, conjunctive query evaluations are generally more expensive and offer less tuning opportunities for a top-k query processor than the typical IR-style "andish" query evaluations that are primarily driven by score aggregations and largely benefit from the dynamic relaxation and compensation of weak matches for some query conditions. However, in order to efficiently prune candidates at an early stage of the query processing in conjunctive mode as well, we aim at testing structural query conditions based on selectivity estimations. In conjunctive mode, all structural conditions in the query, i.e., XPath axes and tag names, as well as the content conditions are mandatory for a document or subtree to qualify for the query result. If a single query condition is missed, the candidate is immediately dropped from the candidate queue and not evaluated any further.

Andish Query Evaluation – XML IR

In order to substantially improve recall as required in most IR-related benchmark settings such as INEX, we can relax this strict notion of conjunctive query evaluations to an "andish" form of content and structure matching which suits most IR-related applications in a much better way. In fact, even for top-k queries with k being set to 10 or 20, Boolean XPath-like query evaluations often yield too few exact matches and thus hinder a conjunctive query processor from providing not only good recall values but also satisfactory precision values for the upper ranks.

In "andish" mode, a result document (or subtree) should still satisfy most structural constraints, but we may tolerate that some tag names or path conditions are not matched. This is useful when queries are posed without much information about the possible and typical tags and paths, e.g., when the XML corpus is a federation of datasets with highly diverse schemata. In the INEX benchmark, a collection of IEEE CS conference and journal publications, the situation arises because of the large number of different

tags and the user's a-priori ignorance about certain content terms occurring in sections or subsections or paragraphs or captions, etc. In this setting our scoring model essentially counts how many structural conditions are satisfied by a result candidate and assigns a small and constant score mass c for every condition that is matched. This structural score mass is combined with the content scores by summation. Note that it is still important to identify non-satisfiable conditions as early and efficiently as possible, because this can reduce the bestscore of a result candidate and make it eligible for pruning. We refer to this "andish" form of query evaluation with relaxable structural and content-related query conditions as the default mode for XML IR.

8.2 Query Decomposition & Index Block-Scans

8.2.1 Query Decomposition & Rewriting

The query rewriter analyzes the query syntax (according to the NEXI or the XPath 2.0 Full-Text specification) and decomposes the query into a number of *navigational* and *content* conditions which are constrained through *structural* conditions each of which is referring to an XPath axis such as the child or the descendant axis. This way, the engine's internal representation of the query is purely graph-based and completely abstracted from the query-language-specific syntax.

Query Trees

Recall from the query language definition in Section 2.2 that the rightmost, top-level node test of a location path is called the *target element* of the query; all other node tests in the location path denote the query's *support elements*. In a strict interpretation of the query structure, only those elements that match the query's target element are considered to be valid results.

Definition 8.2.1 (Navigational Condition) *One or more node tests out of a location path that refer to element nodes only are called navigational conditions.*

For branching path queries, we consider all distinct root-to-leaf paths in the query tree as location paths. Navigational conditions are either inner nodes or leaf nodes of the query tree.

Definition 8.2.2 (Content Condition) *Node tests that refer to text nodes (CDATA) are called content conditions. Each content condition refers to exactly one text token according to the full-content text model for elements.*

Content conditions are always leaf nodes of the query tree. Typically, but not necessarily, the query target elements refer to content conditions.

Definition 8.2.3 (Structural Condition) *The edges of a location path or query tree are called structural conditions. Each edge refers to exactly one XPath axis.*

Navigational query conditions are either single tag conditions using the pre/postorder labeling scheme, or they may encode whole locations paths using DataGuides (see Section 2.3.2 for an example). Content conditions can then be rewritten as combined *locator-term pairs*. For the pre/postorder scheme these are *tag-term pairs*; and for DataGuides, these are *bucketid-term pairs*. A structural query condition constrains the way how the content and navigational conditions are connected. For NEXI, these structural conditions refer to the descendant, self, and attribute axes, only.

Note that depending on the corpus-specific selectivities of the query's location paths, it may be beneficial to use DataGuide-like index structures and bucket ids to encode whole location paths into a single query condition instead of splitting the path into a number of single navigational tag conditions. In both cases, the key idea is to benefit from the lower selectivity of combined navigational and content-related query conditions compared to the selectivities of each tag (or bucket id) and term alone and to reduce the number of random accesses for testing the structure to the largest possible extent. This way, the most important building blocks for querying and ranking, namely the target elements that typically form the leaf nodes of the query, can be efficiently encoded into a combined index for content and structure using tag-term pairs or even bucketid-term pairs for DataGuides. The latter option that dynamically chooses between DataGuides and tag sequences with pre/postorder labels poses additional challenges for queries using the descendant axis and will be discussed in Section 8.5.2. In the following sections, we will focus on the usage of the pre/postorder labeling scheme for the default query processing; the usage of DataGuides follows analogously.

Then the main building block for queries are tag-term pairs that are obtained from merging each term of a query's text node with its nearest preceding tag. Whole location paths with multiple descendant constraints are decomposed into a number of navigational conditions (tag sequences) and combined tag-term pairs for the content-related conditions as leaf nodes in the query tree. Branching path queries can be expressed analogously. Figure 8.4 shows the resulting structure of this merging step for our example query depicted in Figure 8.2. Since the `par` and `title` elements exclusively point to content-related leaf nodes, we do not have to expand the `par` and

`title` tags into separate navigational query conditions but rather merge them with their descendant terms and directly connect the resulting tag-term pairs to the preceding `sec` and `item` element, respectively. This helps us to significantly reduce the dimensionality of the query without changing the semantics of the original query pattern.

Note that we need to introduce an additional edge type for the above query representation to express the self axis for term conditions that have been extracted from the same original text node as shown in Figure 8.4.

Query DAGs

After the query decomposition step, every individual node in the query tree represents either a navigational tag or a combined locator and keyword condition in the form of a tag-term pair. In both cases, the in-memory structural joins are efficiently performed on the pre/postorder information associated with each candidate element in the document tree. For efficient *incremental testing* of structural query conditions and our advanced random access scheduling decisions, we also transitively expand all structural dependencies in the query as illustrated in the step from Figure 8.4 to Figure 8.5. This way, the query forms a directed acyclic graph (DAG) with elementary tag conditions as inner nodes, tag-term conditions as leafs, and all transitively expanded descendant constraints as edges.

This transitive expansion of structural constraints will be key for efficient path validations and random access scheduling and allows for an *incremental testing* of path satisfiability which enables the engine to derive more restrictive bestscore bounds at an early stage of the query processing. This way, the query processor can also take *partial knowledge* about the candidates structure into account without having to evaluate the candidate completely at all query dimensions to provide precise and meaningful worst- and bestscore guarantees.

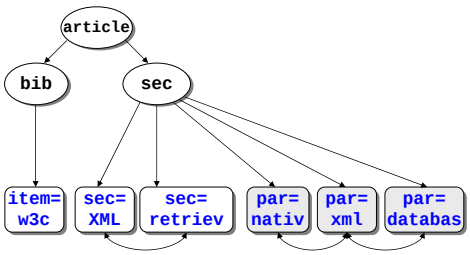


Figure 8.4: Text contents merged with their preceding tags.

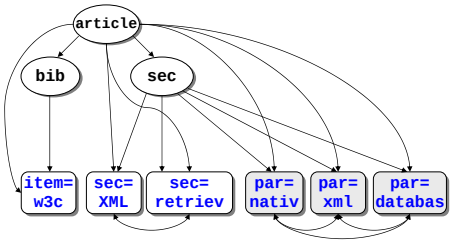


Figure 8.5: Transitively expanded descendant constraints.

For example, in the left branch `//article//bib//item=w3c` in the query DAG of Figure 8.5, an element with tag `item` and content term `"w3c"`) has to be a descendant of both the `article` and the `bib` elements in order to aggregate score mass from the sibling nodes rooted at those elements. If the pre/postorder labels of the `item` element already do not match the pre/postorder labels of the more selective `article` element, we can decrease that element's bestscore by a significant margin (and possibly make it eligible for pruning) without ever having to test the `bib` condition through another expensive random lookup. This is a crucial performance factor for the TopX engine, since it may already prune a major number of items based on their bestscore assumptions and save a substantial amount of index access costs.

8.2.2 Schema Mapping & Index Structures

All inverted index lists are stored in a relational schema; Figure A.1.2 of the Appendix lists the corresponding table and index definitions. An XML element is identified by the combination of the document identifier *docid* and the element's preorder label *pre*. Navigation along all XPath axes is supported by both the *pre* and *post* attributes using the indexing technique by [Gru02], in order to implement a combined inverted index for XML content and structure in a compact way.

All index structures could be implemented as a customized index just as well, e.g., using inverted files with some potential gains in the flexibility of our storage management, in particular trough saving redundant key prefixes in a more object-oriented way than a strictly relational system would allow. In using the given database infrastructure, we basically rely on 30 years of database research for efficient index structures and transaction management (including index updates). In the following, we discuss the detailed schema setup using the *Oracle* database system with the option of leveraging space-efficient *Index Only Tables* (IOTs) [IOT] and the *index key compression* feature for our primary storage structures; all schema definitions can be transfered analogously to other DBMS or inverted files:

- `TagTermFeaturesRA`: IOT with attributes concatenated in the order *(docid, tag, term, score, maxscore, pre, post)* as key.
- `TagTermFeaturesSA`: B^+-tree index over all attributes of `TagTermFeaturesRA` but concatenated in the order *(tag, term, maxscore, docid, score, pre, post)* as key.
- `TagsRA`: Separate IOT with attributes concatenated in the order *(docid, tag, pre, post)* as key.

A sequential scan through an inverted index list now corresponds to an index range scan on the B$^+$-tree index `TagTermFeaturesSA` using the key prefix $(tag, term)$. Sorted access in descending order of scores is guaranteed by the key suffix $(maxscore, docid, score, pre, post)$, where $maxscore$ is the maximum $score$ value among the records grouped by $(docid, tag, term)$, and pre and $post$ denote the position of each tag-term pair in the document tree. By keeping all query-relevant attributes redundantly in the `TagTermFeaturesRA` and `TagTermFeaturesSA` index (and thus forcing a full replication of the index data), we prevent the DBMS from performing more expensive index-access-per-rowid plans (i.e., hidden random accesses between the index and the base table) and are able to perform truly sequential disk accesses on top of the DBMS by reading the all tuples directly from the B$^+$-tree leaf nodes in sorted order. Content-related query conditions for a given document can also be efficiently tested through random accesses on the `TagTermFeaturesRA` IOT (i.e., small index-range-scans and index-skip-scans) using the key prefix $(docid, tag, term)$.

As an additional enhancement, we use Oracle's index key compression option to automatically truncate redundant index key prefixes and skip dispensable key prefix replications at the inner index nodes of the B$^+$-tree structures. This is the case for the $(docid, tag, term)$ prefixes of the *TagTermFeaturesRA* IOT, the $(tag, term, maxscore)$ prefixes of the `TagTermFeaturesSA` index, and the $(docid, tag)$ prefixes of the `TagsRA` IOT which are kept only once in the respective B$^+$-tree index structure and directly link to the actual data instances as leaf nodes of the B$^+$-tree. We fully precompute and materialize the `TagTermFeaturesRA` base table to efficiently support our full-content scoring model on the descendant axis between tags and their succeeding terms, i.e., for patterns of the type `A[.//"a"]` (see Section 3.4.1).

Since elementary tag conditions are defined to contribute to a candidate's aggregated score with a constant score mass c, only, sorted access to those tag conditions would not be beneficial. These structural conditions (i.e., the tags `article`, `bib` and `sec` in the example query) are tested through random access on the `TagsRA` IOT, only.

Document Model

Note that plain text indexing can now be reduced to a special case of the generic XML schema and indexing technique above, using a single virtual and document-wide element with an empty tag (or the wildcard tag `*`) and a pre- and postorder label $pre = post = 1$ for indexing the text document.

On the other hand, the scoring model applied for the XML document's root element (see Section 2.1) refers to the very same scoring model that

a text indexing technique would generate for an unstructured document with the same content. Moreover, all nested elements are treated as mini-documents, each of which is an eligible retrieval unit using their descendant full-contents for scoring and retrieval.

Document Granularity

Keeping the *docid* attribute in our inverted lists basically helps us to quickly identify the hash targets for the inverted block-scans and keeps the range of our in-memory structural joins small and efficient, since path matches are only allowed within the same document and scores are aggregated on a path within a document's boundaries. However, this approach assumes a reasonable document size for *efficient* top-k-style query processing as it is typically given in current benchmark settings such as INEX.

Focusing on XML from a pure data-centric point-of-view, an XML collection might consist of a single large document tree of several Megabytes or even Gigabytes size. This would degrade the performance of the TopX engine, since we would effectively end up performing full scans on the inverted index structures, because every element block would be mapped to the same document. In this case, the TopX algorithm would degenerate to a DBMS-style join-then-sort approach on the TopX index structures. However, it would still allow us to identify and score individual target elements within that large document tree which can be returned as a ranked result list of target elements just as well. Note that even in this case, we benefit from indexing combined tag-term pairs and shorter inverted lists compared to many standard indexing techniques. Moreover, various partitioning approaches have been proposed in the literature to automatically identify and segment large XML trees based on some notion of coherent information units or *index nodes* [FG01] which might be applied as an ad-hoc patch here.

8.2.3 Sorted Access for Element Blocks

Using our full-content indexing approach described in Section 2.1.4, each tag-term pattern can now be mapped onto exactly one of the inverted index lists which corresponds to a strict interpretation of the query's target element in the INEX notation 9.5.1. This way, the data instances of all tag-term pairs are read directly off the inverted lists using efficient sequential accesses. For each such content condition (e.g., `par=native` in our example query), such a sorted index scan is opened by issuing an SQL statement of the form as depicted in Figure 8.6 and incrementally moving a database cursor through the inverted block index for that content condition.

```
select
  score, pre, post
from
  TagTermFeaturesRA
where
  tag = 'item' and term='w3c'
order by
  maxscore desc, docid, score desc
```

Figure 8.6: Index range scan on `TagTermFeaturesRA` as sorted access statement (using the `TagTermFeaturesSA` B^+-tree index) for a given tag and term.

The base table `TagTermFeaturesRA` contains the textual contents encoded as one row per tag-term pair per document, together with the local scores and the pre- and postorder labels. For each tag-term pair, we also provide the maximum score *maxscore* among all entries grouped by document, tag, and term in order to rearrange the tuples in the inverted lists in a convenient way that supports our approach for incremental structural joins to the best possible extent. We thus extend the traditional notion of single-row sorted accesses to a notion of *sorted block-scans*. To efficiently fetch all tag-term pairs connected to a content-related query condition (i.e., a tag-term pair), we perform these sequential block-scans that fetch all the elements per document that are relevant for such a content condition from the respective inverted list through an inexpensive series of sequential disk I/O. The `TagTermFeaturesSA` that materializes this block grouping of elements in descending order of $(maxscore, docid, score)$ for each tag-term condition helps us to this end. Instead of using the tag-term-specific *score* attribute for sorting, we perform sorted accesses for a given $(tag, term)$ key prefix primarily in descending order of *maxscore* using the concatenation of $(maxscore, docid, score, pre, post)$ as key suffix.

The sequential scans now prefetch all tag-term pairs for the same document in "one shot" and keep them in memory for further processing. This allows us to read all the relevant element matches per document by continuously moving a single cursor on each of the physical index list and read them directly from the leafs of the B^+-tree.

If a document is tested at a respective dimension against a path condition and none of its elements meets the path expression, or if a path connection is interrupted at such an element block, we can significantly reduce the bestscore of the whole document and potentially drop the document (and with all its potential target elements) from the queue and the cache. So we can drop many candidates before actually having evaluated them at all

query dimensions and save expensive random IO's compared to existing approaches (e.g., see [KKNR04]), keep the candidate queue smaller and, thus, stop earlier.

Modified Bestscore Bounds

The way we block-scan documents in index lists (or actually their element sets for a given tag-term pair) also affects the way we estimate the bestscores of the current candidates as the *score* attribute would not ensure a monotonous decrease of the $high_i$ values any more. Therefore, in analogy to the single-row sorted accesses model, the *maxscore* attribute now yields the basis for the $high_i$ values used in each inverted list L_i to estimate the upper bestscore bounds for all candidates. Note that the $high_i$ values now serve as a more generous upper bound for the score each candidate can still achieve, because the structural join might actually take an element with a much lower *score* value than the *maxscore* into the path aggregation, but our experiments indicate that pruning remains effective.

We can now update the intermediate $[worstscore(d), bestscore(d)]$ interval of a candidate document d each after having block-scanned d at a respective index list L_i and add the query dimension i to the set of evaluated dimensions $E(d)$, thus excluding uncertainty about further matches of yet unseen elements of d in L_i that – though yielding a lower local score – could potentially yield a higher aggregated path score and might render the score updates non-monotonous. Hence, we will determine the new score bounds of d on the basis of the structural joins between whole element blocks according to the materialized grouping of elements by candidate document and query dimension in our inverted block-index.

Similarly to the text retrieval case, we add the document to the current top-k results, if its worstscore is raised above the current *min-k* threshold. If otherwise its bestscore drops below the *min-k* threshold, we may safely prune d from the candidate queue and the cache. Also note that we can optionally invoke score predictors on the basis of these modified $high_i$ and $bestscore(d)$ values for probabilistic pruning at this point, in order to prune weak candidates similarly to the text retrieval case.

8.2.4 Random Access for Element Blocks

Random accesses to content scores for a given document, tag, and term are performed through additional index range scans on the `TagTermFeaturesRA` index using the triplet $(docid, tag, term)$ as key which is shown in Figure 8.7.

The structure of these statements is precompiled and uploaded to the DBMS to accelerate their executions.

```
select
  score, pre, post
from
  TagTermFeaturesRA
where
  docid = '12345' and tag = 'item' and term='w3c'
order by score desc
```

Figure 8.7: Index range scan on `TagTermFeaturesRA` IOT as random access statement for a given document, tag, and term.

Note that structural tests to the element directory, namely the `TagsRA` IOT, are performed through random access, only, using similar SQL statements. The only difference is that, since the `TagsRA` index merely serves as an element directory to test structural conditions which are encoded as single tags, we are using the concatenation of $(docid, tag)$ as key.

8.3 Structure-aware Top-k Query Processing

Recall from the Section 8.1.2 that the structure-aware path evaluation of candidate poses specific requirements to a top-k query processor. In particular, we are aiming to provide

1) deterministic and monotonous score updates which are independent of the schedule,

2) tight and accurate, quickly converging worst- and bestscore bounds for effective candidate pruning, and

3) incremental score updates taking also partial information about the content and structure into account.

Algorithm 9 shows the basic TopX workflow for processing XML data and structured queries as a very straightforward extension to our previous top-k query processing strategies. Once we have a document's complete set of elements that match a given content condition in memory, we can compare this set against other element sets from the same document, namely, those that we have found through sorted block-scans or random range-scans on the

Algorithm 9 Basic TopX Workflow for Structured Queries.

1: Parse the XPath or NEXI query.
2: Translate query into internal tree representation for path evaluations.
3: Translate query tree into DAG structure which will be used for RA scheduling only.
4: Open cursors to inverted lists for sequential scans on all tag-term leaf nodes.
5: Initialize top-k list, candidate queue, and hash-based cache.
6: Fetch next element block from sequential block-scan according to basic SA scheduling.
7: Join this element block on *did* attribute with other element blocks seen so far.
8: Evaluate path structure for all elements contained in a target element block.
9: Update worst- and bestscore bounds for current document.
10: Determine Min-Probing RA scheduling decisions for expensive predicates.
11: Determine cost-based RA scheduling decisions for remaining content conditions.
12: Determine pruning decisions.
13: Test for *min-k* threshold termination.
14: Return top-k results or continue with step 6.

other index lists for further query conditions. At this point, we compare element sets for the same document against each other, testing path conditions and aggregating local scores. This is performed efficiently using in-memory hash and Staircase [GvKT03] joins on the pre- and postorder labels. Element sets for documents that have at least one element that satisfies all path conditions that can be tested so far are kept around in the cache for later testing of additional path conditions as candidates for further content conditions are fetched, all other elements can be pruned to safe valuable main memory.

8.3.1 TopX Query Processing by Example

For an illustration of the indexing and query evaluation process, consider the example data in Figure 8.8 and the following simple twig query:

`//A[//B[.//,“b”]]//C[.//,“c”]`

(omitting the NEXI-style `about` operator for short). Note that in order to provide a full turn on the algorithm from indexing to querying, we will temporarily switch to a more abstract example with abbreviated tag and term names. All documents are parsed using our full-content scoring model described on Section 3.4.1 and indexed using the inverted block-index over tag-term pairs from Section 2.3.3. Note that this is the same example figure as depicted in Section 3.4.1 to illustrate the scoring model.

Figure 8.1 shows an excerpt of the respective `TagTermFeaturesSA` B$^+$-tree index. For simplicity, we do not show the Okapi-based scores here, but rather pretend that our scores are mere ftf values normalized by the number of terms in a subtree (e.g., for the tag-term pair `A:a`, the element e_{10}

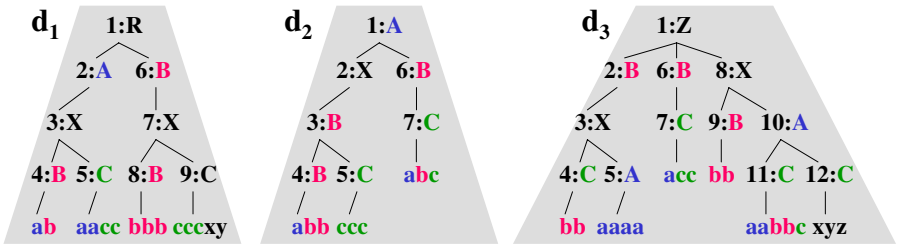


Figure 8.8: Some simple XML example documents.

in document d_3 has score 1/4, because the term a occurs twice among the eight terms under e_{10}).

L_i	tag	term	maxscore	$\mathbf{d}_{id}$	score	pre
1	A	a	1	d3	1	e5
	A	a	1	d3	1/4	e10
	A	a	1/2	d1	1/2	e2
	A	a	2/9	d2	2/9	e1
2	B	b	1	d1	1	e8
	B	b	1	d1	1/2	e4
	B	b	1	d1	3/7	e6
	B	b	1	d3	1	e9
	B	b	1	d3	1/3	e2
	B	b	2/3	d2	2/3	e4
	B	b	2/3	d2	1/3	e3
	B	b	2/3	d2	1/3	e6
3	C	c	1	d2	1	e5
	C	c	1	d2	1/3	e7
	C	c	2/3	d3	2/3	e7
	C	c	2/3	d3	1/5	e11
	C	c	3/5	d1	3/5	e9
	C	c	3/5	d1	1/2	e5

Table 8.1: Block index for the example of Figure 8.8

TopX evaluates the query by opening index scans for the two tag-term pairs `B:b`, and `C:c`, and block-fetches the best document for each of the two conditions. For example, for `B:b`, the first three lines of index list L_2 in Figure 8.1 that belong to the same document d_1 are fetched as a sorted block-scan. Figure 10 shows pseudo code for the TopX algorithm. As the index scans proceed in a baseline round-robin fashion among all index lists L_i connected to content conditions (i.e., tag-term pairs), the algorithm continuously computes $[worstscore(d), bestscore(d)]$ intervals for each candidate d that it is fetched by a sorted access and periodically updates the $bestscore(d)$

values of all candidates using the current $high_i$ values and the score mass of the not yet evaluated structural constraints $o_j \cdot c$.

As the first round of block-scan fetches yields two different documents, the algorithm needs to continue fetching the second-best document for each condition. After the second round, it happens that d_3's relevant elements for both conditions are in memory at this point. A random access for all `A` elements in d_3 can now be triggered, if it is cost-beneficial (see Section 8.4). If so, we can efficiently test both path conditions for d_3, namely whether a `B:b` element is a descendant of an `A` element and a `C:c` element is a descendant of the same `A` element, by comparing the pre- and postorder numbers of the respective element pairs. This way, it is detected that none of d_3's element triples satisfies both path conditions and the $wortscore(d_3)$ and $bestscore(d_3)$ values have both converged to the final value $1 + 2/3$, however, without the score mass $c = 1$ for the missed `A` condition. The same test can be performed for document d_1 at this point, but only for one of the two path conditions, namely, whether an `A` element has a `B:b` element among its descendants. The second condition, namely the connection between `A` and `C:c`, can be tested only later, when the matches for `C:c` within d_1 are encountered on the `C:c` index list. As d_1 has valid element pairs after the `A` vs. `B:b` test, we recompute the $[worstscore(d_1), bestscore(d_1)]$ interval which now becomes $[1 + 1, 1 + 1 + 3/5]$. If $worstscore(d_1) > min\text{-}k$, we put d_1 into the top-k results; if otherwise the $bestscore(d_1) > min\text{-}k$, we put d_1 into the candidate queue and reserve the option to perform the probabilistic threshold test as to whether d_1 still has a good chance to qualify for the top-k.

The following sections describe the algorithmic details for these evaluation strategies.

8.3.2 In-Memory Structural Joins

Algorithm 10 shows only a minor modification compared to the core TopX query processor described in Section 4.3, namely at the point where a candidate's worstscore is determined. For XML document retrieval, the document's final score is defined as the maximum path score in d's element structure that matches the query's target element, i.e.,

$$worstscore(d) := \max\{getElementScore(d, targetNode, e) | e \in targetElements(d)\}$$

where $targetNode$ denotes the query target node and $targetElements(d)$ refers to the element block that is fetched for d from the inverted block index and matches $targetNode$. Note that in order to ensure that we only retrieve

Algorithm 10 Structure-Aware Index List Processing.

```
1:  processIndexListXML(Index List L_i, Batch-size b_i)
2:  isAlive_i = true;
3:  isSuspended_i = false;
4:  pos_i = 0;
5:  while isAlive & L_i.hasNext() do
6:    // Scan for next element block in L_i
7:    <docid,maxscore,elements(d,i)> = L_i.getNextBlock();
8:    d := cache.getCachedItem(docid);
9:    E(d) := E(d) ∪ {i};
10:   high_i := α_i(β_i+maxscore);
11:   pos_i++;
12:   // Update worst- and bestscore bounds
13:   worstscore(d) :=
        max{ getElementScore(d,targetNode,e) for all e∈ targetElements(d) };
14:   bestscore(d) := worstscore(d) + Σ_{ν∈Ē(d)} high_ν;
15:   // Continue as in core query processing algorithm shown in Algorithm 4
16:   ...
17:   if GRANULARITY = ELEMENTS then
18:     min-k := getTopkElements(top-k)[k];
19:   else
20:     min-k := top-k.getMinkScore();
21:   end if
22:   ...
23: end while
```

valid target elements as results, it is cheapest to start the recursive path traversal only for elements at the query's target node rather than maximizing path scores for all possible element combinations and then filtering those that actually match the target node condition.

Thus, for a given query target node and each element in a block associated with that target node, we invoke a recursive tree traversal using the *getElementScore* procedure (see Algorithm 11) to determine the maximum path score among all target elements seen so far, which will then be used to determine the document's worstscore. These in-memory structural joins and path evaluations for a candidate d are updated *incrementally* after each sequential block-scan on d on a different tag-term index list, i.e., whenever we gain additional information about a candidate document's element structure. All the remaining building blocks of the TopX query processor such as the candidate and top-k bookkeeping remain unchanged.

Document vs. Element Retrieval

Algorithm 10 also has an option that distinguishes between document and element granularity for retrieval. In document mode, we use the worstscore of the rank-k document of the current top-k document list to determine

the *min-k* threshold as before; in element mode, we use the score of the rank-k element among the current top-k documents to determine the min-k threshold. Note that the rank-k document score is a lower bound for the rank-k element score. We will revisit this issue in Section 8.3.6.

8.3.3 Incremental Path Tests

The focus of our algorithm lies on the efficient evaluation of path queries over partially evaluated candidate documents, thus dealing with uncertainty about both content-related and structural query conditions. We introduce a novel approach for incremental path testing that comprises a combination of efficient hash-joins for content-related query conditions and Staircase joins (from the XPath Accelerator work [Gru02, GvKT03]) for the structure. We extend this approach by a notion of *virtual elements* with 0 scores that enable the engine to also navigate through the structure of partially evaluated candidate documents.

Figure 8.9 first illustrates the algorithm for a fully evaluated candidate document d and the example query from Table 8.1, we will then extend the approach for partially evaluated candidates. The figure shows all element blocks for d depicted as score, pre- and postorder triplets of the form $score[pre, post]$ for all element blocks that have been mapped to the individual nodes of the query pattern. This is the case when all query conditions have been successfully tested on d by a combination of sorted and random accesses to our inverted block index as described in Section 2.3.3. In the following, we denote the element block associated with a query node n at document d as $elements(d, n)$. For example, the rightmost element block in Figure 8.9

`par=databas`

0.071 [389, 388]
0.068 [354, 353]
0.041 [375, 378]
0.022 [372, 371]

refers to the element block of candidate d for the target content condition `par=databas` (including stemming) of the example query of Figures 8.1 to 8.5. Each of the entries represents a distinct element of the candidate document d. The bold-face element entry ***0.068 [354, 353]*** refers to the `par` element with the preorder label *354* matching the term "databas" among its full-contents that will maximize the local content conditions for the target node and in fact aggregate the highest path score with regard to the whole query. Hence, it will determine the document's final score.

Then query evaluation and recursive tree traversal is performed along the node structure of the query, with individual elements $e \in elements(d, n)$ being joined at each query node n for score aggregation. Note that – although d yields valid local matches for each of the query conditions – the query might still not be satisfiable by d in a conjunctive sense, since at least one element combination has to lie on a connected path structure that matches the whole query pattern. Since only those elements that are specified as target elements by the query are defined to be valid result elements and to obtain a non-zero score, we have to *start evaluating* the candidate at the elements matching a *target query condition* which corresponds to the `par` node in our example. Starting with these targets, we traverse the query tree in two opposite directions (only considering the query tree structure without the transitively expanded descendant edges at this point) to make sure we start with a valid result element. For each of the target elements, we aim at *maximizing* the aggregated score of a connected path from a target leaf, over its parent nodes and down to its valid siblings. The top-scored target element finally yields the document's score.

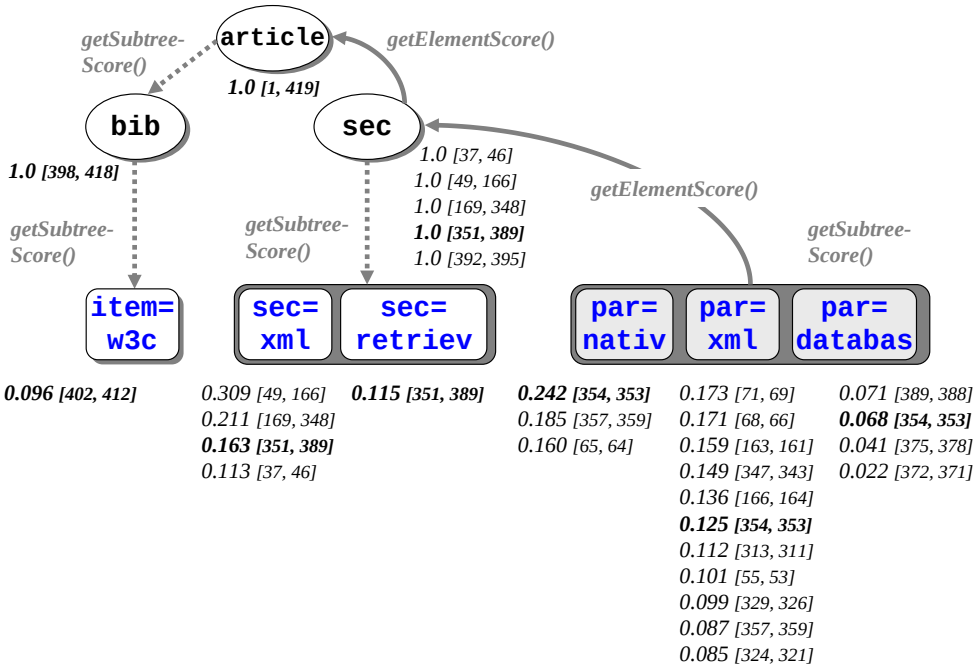


Figure 8.9: Path query evaluation on a candidate's element block structure for the example query of Figure 8.2.

The *getElementScore* Procedure

For a given target element and query target node, the main procedure *getElementScore*, see Algorithm 11, iteratively traverses the query tree "upwards" all the target elements' parents up to the query root node simply by inverting the direction of the descendant constraints. We choose the target node of the query as our *initial context node* for the path evaluation. Each of the entries of the parent's element block is joined with the source element by comparing the pre- and postorder labels, see Algorithm 11. Note that all query conditions that relate to the query subtree rooted at the query node of the previous iteration are excluded from the next round of the *getSubtreeScore* aggregation in order to avoid redundant score aggregations. At each level of the query tree, we choose the element that *maximizes* the aggregated score of the *getSubtreeScore* and *getParentScore* calls, respectively, in the case that multiple elements qualify for a query subcondition based on their pre- and postorder labels.

Algorithm 11 Iteratively navigate "upwards" the query tree to the root.

```
 1: getElementScore(Document d, Query node n, Element e):
 2: // Recursively retrieve the score of the subtree rooted at the parent node
 3: // each excluding the previously retrieved subtree; stop at the query root node
 4: score := 0;
 5: contextNode := n;
 6: previousNode := null;
 7: while contextNode ≠ null do
 8:    subtreeScore :=
         max{getSubTreeScore(d, contextNode, previousNode, e) |
           e.pre < sourceElement.pre & e.post > sourceElement.post
               for all e∈elements(d, contextNode)
         };
 9:    // If one subtree recursion yields a zero score in conjunctive mode
10:    // then break the evaluation of the current element
11:    if MODE == CONJUNCTIVE & subtreeScore == 0 then
12:       return 0;
13:    end if
14:    score += subtreeScore;
15:    previousNode := contextNode;
16:    currentNode := contextNode.parent;
17: end while
18: return score;
```

The *getSubtreeScore* Procedure

Starting from an arbitrary source element and query node, the recursive procedure *getSubtreeScore*, see Algorithm 12, queries the candidate's element block structure for all elements that are connected to the source element

with regard to the self and the descendant-axis by comparing pre- and post-order labels. Query leaf nodes that refer to content-related query conditions and which are connected by a self-constraint (denoted by a dark background rectangle in Figure 8.9), are hash-joined on the preorder attribute and cumulate their local scores to an aggregated element score for the respective query subcondition. Again, for structural query conditions, i.e., inner nodes of the query tree, the method recurses into all the element blocks for the query's subtree structure according to the descendant constraints now based on the comparison of both the pre- and the postorder labels, see Algorithm 12. Note that, since we always start traversing the query at the its target dimension, each recursion finally contributes to the aggregated score of one of the target elements for a given candidate document.

Algorithm 12 Recursively retrieve the aggregated path score for a given element at a query subtree.

```
 1: getSubtreeScore(Document d, Query node n, Excluded sibling node x, Element e):
 2: contextNode = n;
 3: excludedNode = x;
 4: if contextNode.isLeaf then
 5:   // Hash-join all element blocks related to contextNode using e.pre
 6:   subtreeScore := ∑_{n∈self(contextNode)} {e.score | e.pre = e'.pre for all e'∈elements(d,n)};
 7: else
 8:   // Staircase-join all elements blocks related to children of currentNode
 9:   // using sourceElement.pre and sourceElement.post
10:   subtreeScore := e.score;
11:   for all i = 1..#currentNode.children do
12:     if contextNode.child[i] != excludedNode then
13:       subtreeScore +=
            max{getSubTreeScore(d, contextNode.child[i], null, e) |
              e.pre < e'.pre & e.post > e'.post for all e'∈ elements(d, contextNode.child[i])
            };
14:     end if
15:   end for
16: end if
17: return subtreeScore;
```

In the example structure of Figure 8.9, we initialize the algorithm by first hash-joining all element blocks for the three query conditions that refer to the target *par* element, namely the `par=native`, `par=xml` and `par=databas` content-conditions. We see that the `par=native` condition has a relatively low selectivity with 3 matching elements for that candidate; `par=xml` has the highest selectivity with 11 matches; and `par=databas` has 4 matches. After hash-joining all three element blocks for that query target dimension, there are still 15 distinct target elements left, each of which is already a valid match for the query. For each of these, we have to start a *getElementScore* recursion for the remaining query dimensions and scores. For simplicity, we only consider the element with the preorder label 354 (emphasized in

bold face) here, which yields the best aggregated score of 0.435 so far. The parent query condition `sec` yields 5 more elements out of which only the one with the preorder label 351 qualifies for further traversal by its pre- and postorder labels and thus contributes to the aggregated score of element 354 with a value of 1.278 using the *getSubtreeScore* traversal. Similarly, the second iteration of *getElementScore* yields the only `article` root element with a preorder label of 1 and a static local score of 1.0. From here, the *getSubtreeScore* method recursively navigates down two levels to the two `bib` and `item=w3c` query conditions which are also found to provide valid element matches that contribute to the aggregated score of element 354 with values of 1.0 and 0.096, respectively, after checking their pre- and postorder labels. Finally, element 354 obtains an aggregated score of 3.809 which also makes it the top-scored element out of the 15 distinct target elements for the target `par` condition. Note that it is also the only element that satisfies this query in a conjunctive sense.

Conjunctive Mode

Algorithm 11 has an option to terminate an element's evaluation in conjunctive mode, if any query subtree recursion or single query dimension yields a local score of 0 for the path traversal on that candidate. Note that this does not necessarily render the whole candidate invalid as it may still be encountered in all of its remainder dimensions, so we may only assign it a conservative worstscore bound of 0 at this point. Breaking the evaluation in conjunctive mode may be due to the fact that at least one query condition i

1) has not been fully evaluated yet ($\exists\, i \; with \; i \notin E(d)$) through the sequential block-scans, so we do not yet know if the candidate will still satisfy the query conjunctively, and the candidate is kept in the queue

2) has been tested ($i \in E(d)$), e.g., through a random lookup, but the inverted list L_i does not contain any match for that document, and the candidate is dropped, or

3) has been tested ($i \in E(d)$), but there is no valid path from a target element to any of the elements at dimension i based on their pre/postorder labels, and the candidate is dropped.

In all three cases, the document and, thus, all its target elements obtain a $worstscore(d)$ of 0. In the first case, $bestscore(d)$ is assigned a positive value as the document is not yet evaluated at all query dimensions and may still provide a valid path match for all query conditions. Note that we may already

take partial knowledge about the candidate's structure into account in order to provide an as-tight-as-possible $bestscore(d)$ bound. In the latter two cases, the document obtains also a $bestscore(d) = 0$, and thus the evaluation of d terminates, and d can be safely pruned, since one or more query conditions have been tested on d that did not provide a valid path match which is not allowed in conjunctive mode. So the distinction between "andish" and conjunctive mode affects the estimation of both the worst- and bestscore bounds.

Andish Mode

In andish mode, the evaluation of d is not terminated due to a single failed query condition, but $worstscore(d)$ is increased as soon as one of the query's target elements is positively matched against d and it further increases with partially fulfilled path conditions for further support elements that are connected to the target element. Similarly, $bestscore(d)$ is not reset to 0 if a single condition fails, but the algorithm assumes that other element blocks for the remaining query conditions may still contribute to the document's score, even if we cannot match any path starting from a target element in the sense of a Boolean XPath-like evaluation any more.

Unsurprisingly – and in contrast to conventional database queries – conjunctive query evaluations are more expensive to evaluate for a top-k engine than the "andish" counterpart, because the $[worstscore(d), bestscore(d)]$ intervals converge much slower and low-scored content matches cannot be compensated for queries with a drastically reduced conjunctive join selectivity. In the following, we will focus on the "andish" evaluation strategy as the much more interesting but also more difficult case for XML IR involving incremental path validations. The conjunctive mode is merely kept as an option to support Boolean-XPath-like query evaluations as demanded by some applications.

8.3.4 Virtual Navigational Elements

Virtual Support Elements

Now, unfortunately, the situation is not alway as simple as in the above setting, with fully evaluated candidate documents at all query conditions. Since our query evaluation strategy focuses on performing mainly efficient sorted access to the disk-resident inverted index structures, the order in which documents are encountered in the inverted lists by the query processor is unpredictable and virtually random. With only partial knowledge about the

document structure, our structure-aware query processing algorithm would terminate evaluations when the path structure is interrupted at any node in the query tree – and so would do any traditional technique such as Holistic Twig Joins or the original Staircase joins. Moreover, if the query target node has not yet been evaluated, there would not even be an anchor node to start the evaluation process, because the remainder of the candidates element structure would simply not be reachable. This would render the worst- and bestscore bounds overly conservative and bar the top-k query processor from reaching the *min-k* threshold termination comparably early as in the text case, although we might have already gained substantial knowledge about the partial path structure that could be taken into account for scoring.

In order to avoid these situations, we introduce the notion of *virtual support elements* for the inner nodes of the query tree with a local score of 0 and an any-match option for the pre- and postorder-based Staircase joins (thus conceptionally attaching entries of the form *0.0 [*,*]* to each element block). These "joker" elements may be joined in the *getElementScore* and *getSubtreeScore* traversals with any real element match or with other virtual support elements for navigation through unevaluated navigational query conditions. Even after an element-block for an inner node is fetched from disk, we keep the virtual navigational element for that node. This simple tweak prevents us from having to reject the score of a subtree that might have already been granted to the candidate by a previous update using a virtual element for navigation which might render the score updates non-monotonous. This way, the content nodes serve as a synapse for connecting whole query subtrees, without having to necessarily make the actual random lookup for the connecting path condition. In "andish" query evaluation mode, we may safely increase the worstscore of a candidate d without having to assume a connected path structure which let's us provide more accurate worst- and bestscore bounds taking into account *all* the evaluated query conditions. In many cases, the bestscore of a candidate document based on its content-related query conditions might already make it eligible for pruning without having to perform the actual random lookups onto the structure.

The value of the *static score mass* c that every candidate is about to aggregate for a successfully tested structural query condition determines whether we are in favor of matching the query structure or the content-related query conditions among the top-scored result documents. A large value of c tends to dominate the content-related query conditions and will make the algorithm choose only real support elements with a score of c for the maximum path score, although it might have to neglect some lower-scored content conditions that do not match the structure. A low value of c, on the other hand, tends to choose the content-related query conditions for the

aggregated path score and might accept some zero-scored virtual support elements. In the above example (and in the experiments section), we chose $c = 1.0$, i.e., we put a relatedively high emphasis on the query structure (note that content scores are normalized to 1 and most tag-term pairs indeed have a significantly lower score than 1).

Virtual Target Elements

As mentioned above, the lack of a target element in the candidate's path structure would prevent our algorithm from processing a candidate at any of the remaining query dimension and especially keep the bestscore bound unnecessarily high. Similarly to the virtual support elements, which mainly serve to reason more accurately about a candidates worstscore, we also introduce the notion of *virtual target elements* with a local score of 0 and the same any-match option for the pre- and postorder-based Staircase joins.

The only difference between the two kinds is that a virtual target element helps us to more accurately restrict a candidates' bestscore, whereas, according to our scoring model, the worstscore has be reset to zero as long as no valid target element has been detected. Again, both the worst- and bestscore aggregations remain monotonous, because any subsequent update on the candidates structure with a real block of target elements may only raise the worstscore above zero and – at the same time – further decrease the bestscore.

8.3.5 Complexity

Subtree Caching

As described in the previous subsection, element blocks for content-related query conditions are efficiently hash-joined in linear runtime $O(m'b)$ with regard to the maximum block size b and the number of self-constrained content conditions m'. Algorithm 11 shows that, in order to get the top-ranked target element for a given candidate document and thus maximize its worstscore, the parent-nodes are traversed repeatedly for all target elements of that candidate document. Obviously, for complex queries the amount of CPU time spent on recursively evaluating the path structure may be a significant fraction of the overall query processing time and might even dominate the time spent on physical disk I/O.

An effective way of avoiding repeated tree traversals is to introduce an additional level of hashing for the Staircase joins, too, and to *cache* the aggregated scores of all inner element nodes whenever the respective query

subtree is traversed for the first time and to remember the resulting score of the whole *getSubtreeScore* recursion for a given query node and source element.

Runtime Complexity of the Tree Algorithm

If we consider the two procedures *getElementScore* and *getSubtreeScore*, we see that the algorithm performs nested-loop-like Staircase join steps for each two element blocks between an inner node of the query tree and each of its child nodes, with quadratic complexity for each of these Staircase joins. These Staircase joins are performed f times for a node with f children. Note that the amount of elements at all nodes (and hence the maximum block size) remains constant, because we merely aggregate scores for the existing elements, but no new elements are added to any of the nodes. Then the runtime complexity of our in-memory structural joins is $O\left((f\,b^2)^h\right)$, for a maximum query fanout of f (only counting true structural descendant relationships between tags at this point, because content conditions are hash-joined anyway), a maximum element block size of b, and a query tree height of h. Here, the maximum element block size b refers to the document complexity, and k and h refer to the query complexity. Note that although our algorithm is exponential in the query height, this specialized type of path evaluation is designed to suit the typical IR-style NEXI queries and our incremental path tests to the best possible extent, thus dealing with a major amount of content conditions and ensuring monotonous score updates with uncertainty in the structure for partly evaluated candidates.

Although XPath evaluations have been shown to be solvable in polynomial time $O(|D|^5\,|Q|^2)$ with a bottom-up, dynamic programming algorithm [GKP02, GKP03], these approaches hardly apply to our application, since they are assuming conjunctive queries only and cannot easily be adapted to efficiently deal with uncertainty in the structure (similar assumptions apply for Holistic Twig joins [BKS02, CMW03] as well).

We believe that our specific solution is superior in terms of both runtime complexity and scoring flexibility for typical XML IR queries. It largely benefits from a major amount of efficiently hash-joined content conditions and queries that hardly ever exhibit a query tree height of more than 2. Note that this is a very pessimistic upper bound, as is does not take the subtree caching for the Staircase join into account. With this option enabled, the complete tree structure is fully traversed only for the first target element out of the target element block of the query. After that, all parent and descendant scores at inner query nodes are cached, such that each support element is touched at most once per candidate update.

8.3.6 Element Retrieval

With the above building blocks, we are able to retain the very same top-k and candidate bookkeeping strategies developed for document retrieval also for element retrieval. Returning the *top-k target elements* instead of the top-k documents, is very straightforward with our approach. Since, in the beginning of this section, we defined the score of a document to be the maximum score of a path from a target element across its parents and down to its siblings. Then the document's worstscore is an upper bound of all target elements' worstscore; and so is the document's bestscore an upper bound of all target elements' bestscore. Now we can easily extend our top-k retrieval granularity from returning the top-k documents to returning the top-k elements just by extracting further target elements from the current top-k documents with scores that are less or equal to the document score. Hence, the kth-ranked element yields a new $\widetilde{min\text{-}k}$ threshold condition that is equal or greater than the kth-ranked document which is now a *true top-k algorithm for XML element retrieval.* It even allows us to dynamically switch between document and element granularity with almost no computational overhead. We extend the existing algorithm by

1) efficiently merging the target elements of the current top-k documents,

2) introducing a new main-memory top-k-style breaking condition for this merge, and

3) refining the $\widetilde{min\text{-}k}$ threshold for algorithm termination now based on the kth-ranked target element.

Adaptive Thresholds for the Top-k Elements

Algorithm 13 shows an efficient merging of k *sorted linked lists* of target elements as an in-memory top-k algorithm for this merging step. In lines 8.3.6 and 8.3.6, it has two break conditions for the merging procedure for the target elements obtained from the current top-k document, namely whenever the score of the current element list is below the original document-based *min-k* threshold. The efficient merging of target elements from the top-k documents has a crucial impact on query runtime, since it is an internal loop of the TopX query processor. It is called iteratively whenever thread synchronization is done and the min-kstopping condition is tested.

Note that using element granularity even helps to further increase retrieval efficiency, because the increased threshold $\widetilde{min\text{-}k} \geq min\text{-}k$ leads to

more aggressive candidate pruning and earlier threshold termination. As an obvious example, consider the query

```
//article[about(.//sec,‘‘Native XML Databases’’]//bib//item
```

where the `item` tag denotes the query's target node. Processing this query at element granularity for the top-10 target elements effectively means we are looking for the top-1 document that matches the respective content conditions best. For this top-ranked document, we just have to schedule a few random range scans on the `TagsRA` IOT for the tags `bib` and `item` to return all bibliographic citations under the `//article//bib//item` path which will most probably be more than 10 target elements for the top-1 document ranked after the scores for "Native XML Databases" already (for the typical IEEE journal or conference proceedings from the INEX collection). Since all target elements are joined with the aggregated score for these content conditions under their common `article` root, they all get perfectly the same final score.

8.4 Random Access Scheduling for Structural Conditions

The rationale of TopX is to postpone expensive random accesses as much as possible and perform them only for the best top-k candidates. However, it can be beneficial to test path conditions earlier, namely, for eliminating candidates that do not satisfy the conditions but have high worstscores. Moreover, in the query model where a violated path condition leads to a score penalty, positively testing a path condition increases the worstscore of a candidate, thus potentially improving the *min-k* threshold and leading to increased pruning subsequently. In TopX, we consider random accesses at specific points only, namely, whenever the priority queue is rebuilt. At this point, we consider each candidate d and decide whether we should make random accesses to test unresolved path conditions, or look up missing scores for content conditions.

For this XML-specific scheduling decision, we have developed two different strategies. The first, coined *Min-Probing*, transfers the expensive predicate paradigm to the XML case. Since the structure of a document contributes a constant score mass c for each matched structural query condition, these structural tests are performed through random lookups to the `TagsRA` index only, i.e., they cannot be resolved through sorted accesses at all. The Min-Probing scheduler tests all candidates that are about to be

promoted to the intermediate top-k results individually on the basis of their known worstscores, indifferently of their estimated scores or the selectivity of the remaining structural query conditions. It is a simple and safe approach that guarantees all top-k items to have only fully resolved path conditions at any time of the algorithm, but it is not cost-aware and potentially schedules too many lookups onto candidates, if the structured parts – and moreover the not yet resolved support elements – have a very low selectivity.

The second, coined *Ben-Probing*, extends the cost model developed in Section 6.4.2 from the unstructured case to structural query conditions in XML retrieval. This cost model can be combined with the content-based scheduler to trigger lookups to both structural and content-related query conditions for a given candidate. This model is cost-aware and aims at a balanced amount of SA and RA costs, taking the c_R/c_S cost ratio between random and sorted accesses into account. The tricky thing is again the appropriate *probabilistic ordering of candidates* for the random lookups with regard to their estimated scores and the combined content-related and structural selectivities of query conditions that they are still about to match.

8.4.1 Min-Probing

Min-Probing aims at a minimum number of random accesses by probing structural conditions for the most promising candidates, only. Since we do not perform sorted access on structural query conditions such as tag sequences and branching path conditions, we adopt the notion of *expensive predicates* in the sense of [CwH02] for these navigational query conditions. We schedule random accesses only for those candidates d whose

$$worstscore(d) + o_j \cdot c > min\text{-}k \ , \tag{8.1}$$

where o_j is the number of untested structural conditions for d and c is a static score mass that d earns with every satisfied structural condition according to our scoring model (see Section 3.4.2). Then $o_j \cdot c$ corresponds to the expensive predicate deficit $gap(d)$ that d would accumulate for the matched predicates in addition to its already known worstscore as defined in Section 4.2.

This way, we schedule a whole batch of random lookups, if d has a sufficiently high $worstscore(d)$ to get promoted to the top-k when the structural conditions can be satisfied as well. If otherwise $bestscore(d)$ becomes less than the current $min\text{-}k$ threshold after the random lookups, we may safely drop the candidate. This has the positive effect that the top-k index list only contains items whose structure has already been verified and proven to yield a sufficiently high score mass to push the candidate into the current top-k list (with a respective boost for the $min\text{-}k$ threshold as well).

This scheduling strategy suits best for a combination of lowly selective (i.e., infrequent) tag-term pairs as query targets and highly selective (i.e., frequent) support elements, that is, the ranking is mostly decided by the content-related query conditions (the tag-term pairs, or query leafs), and the random accesses for the structure (the inner nodes that connect the tag-term pairs) do not fail often. In the experiments in Section 9.8, we will see that in fact most INEX queries are very close to this kind.

The Min-Probe approach also fits with any content-based SA an RA scheduling approach as described in Sections 6.3 and 6.4. It is lightweight and inexpensive to evaluate, such that it can be tested after each sorted block-access and for each candidate that is about to be promoted to the top-k results. It is therefore part of the core TopX query processor, see Section 4.3.

8.4.2 Ben-Probing

Ben-Probing uses an analytic cost model that is closely related to the one introduced in Section 6.4.2 for plain inverted lists, e.g., in the text retrieval case. We assume that there are o_j still unresolved structural query conditions that relate to at least one path or twig constraint for a candidate d. The total number of index entries still to be scanned in the ith index list is denoted by l_i and the overlap of lists in the presence of explicit correlation statistics is denoted as l_{ij}, respectively. We denote the number of documents in the priority queue by $|Q|$, and the batch size for the next round of sorted accesses on the index lists by $b = \sum_i^m b_i$. The probability that document d, which has been seen in the tag-term index lists $E(d)$ and has not yet been encountered in lists $\bar{E}(d) = [1..m] - E(d)$, qualifies for the final top-k result is estimated by the combined score predictor and selectivity estimator similarly to the random access scheduling approach for plain inverted lists (see Section 6.2.2) and denoted as $p(d)$. Notice that the histogram-based score predictors (or closed-form score estimators) are now based on precomputed statistics about the *joint tag-term* score distributions for the full-content scoring model and our inverted index structure over tag-term pairs. Selectivities are further determined at document level, e.g, a value of $s_{11} = 0.055$ for the tag-term pair `p=xml` (see Table 8.2) means that 5.5 percent of the documents in the corpus contain that particular tag-term pair. This approach translates our previous scheduling and probabilistic pruning assumptions for plain inverted lists directly to the XML case and helps granting compatibility among the different software components. To adopt these existing approaches toward a true structure-aware scheduler, we now refine the selectivity estimator to also take characteristic XML patterns into account.

Selectivity Estimators for Structural Query Conditions

In order to adopt our previous selectivity estimations assumptions, we break down the query structure into the following basic structural query patterns:

- *Tag-Term Pairs* – Merged tag-term pairs for content-related conditions.
- *Descendants* – Tag pairs for transitively expanded descendant relations.
- *Twigs* – Tag triplets of branching path element for transitively expanded descendant relations.

We now estimate the selectivity of the o_j remaining structural query conditions that are connected to one or more path or twig patterns by precomputed corpus frequencies of ancestor-descendant and branching path elements, i.e., pairs and triples of tags. Note that is a very simple form of XML synopsis; it could be replaced by more advanced approaches like [AAN01, LWP+02, WPJ03], but our experiments indicate that this approach already yields a very effective method for pruning and identifying *which* candidate and also *when* it should be tested by an explicit random lookup on the structure.

```
//article[
  //sec[
    about(.//, "XML retrieval")
    and
    about(.//bib, "W3C")
  ]
]//par[
 about(.//,"native XML databases")
]
```

Figure 8.10: Slightly modified NEXI query with a very lowly selective structural condition.

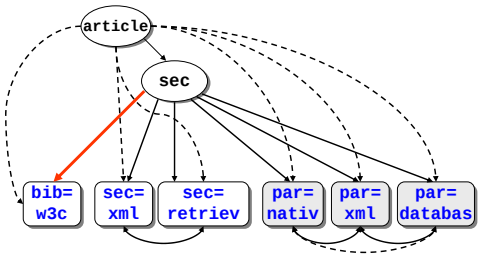


Figure 8.11: DAG structure of the modified query.

Each query condition – or node in the query DAG – can now be associated with one or more of these structural query patterns. For example, the content-related tag-term condition `par=xml` is connected to the two paths `article//par` and `sec//par` and to the twig `article[//bib]//par`. Note that we disregard element ordering and thus consider symmetric patterns only, e.g., the twig `article[//bib]//par` is equivalent to the twig `article[//par]//bib` and is therefore assigned the same selectivity. The

non-symmetric case would follow directly by extracting more detailed statistics for order-aware tag patterns if required.

In order to take a structure-specific scheduling decision, we always consider the *largest subset* of unresolved structural path and twigs patterns for a candidate that are non-overlapping in their structure. For example, if we have the three unresolved query conditions `sec`, `bib=w3c` and `p=xml` for a partially evaluated candidate, we do not multiply the selectivities $s_5 = 0.002$ and $s_6 = 0.964$ of the two connected paths `sec//bib` and `sec//p` *and* the twig selectivity $s_3 = 0.002$ for the `sec[//bib]//par` twig, but rather use the twig selectivity $s_3 = 0.002$ as combined selectivity estimate alone.

Type	**Pattern**	**Selectivity**
Twigs	`article[//bib]//par`	$s_1 = 0.682$
	`article[//bib]//sec`	$s_2 = 0.613$
	`sec[//bib]//par`	$s_3 = 0.002$
Descendants	`article//bib`	$s_4 = 0.614$
	`article//par`	$s_3 = 0.982$
	`article//sec`	$s_4 = 0.982$
	`sec//bib`	$s_5 = 0.002$
	`sec//par`	$s_6 = 0.964$
Tag-Term Pairs	`bib=w3c`	$s_7 = 0.007$
	`sec=xml`	$s_8 = 0.055$
	`sec=retriev`	$s_9 = 0.171$
	`par=nativ`	$s_{10} = 0.044$
	`par=xml`	$s_{11} = 0.055$
	`par=databas`	$s_{12} = 0.319$

Table 8.2: Basic structural query patterns and their selectivities for the example twig query of Figure 8.10.

For the example query in Figure 8.11, we intentionally switched the query structure compared to the initial example query of Figure 8.2, such that we now require the `bib` element not to be a descendant of the `article` element any more but a descendant of a `sec` element. This infringes upon the INEX schema convention, however the selectivity of that path condition `sec//bib` is indeed low but not 0; Table 8.2 shows the actual values for the INEX collection. Recall that we do explicitly support such queries and try to optimize our processing strategies for such situations, since we are aiming at schema-oblivious and non-validating indexing and querying. Any sequence of random accesses to a partially resolved candidate d would now start with the two query conditions for this least selective edge, namely `sec//bib`, and cancel further lookups on d, if because of the first random I/O $bestscore(d) \leq$ *min-k* and d could be pruned already.

Cost Model for Random Lookups to Structural Conditions

BenProbe compares the cost of making random accesses to tag-term index lists or to indexes for structural path conditions versus the cost of proceeding with the sorted-access index scans. For all three cost categories, we consider only the *expected wasted cost (EWC)* which is the expected number of random (or sorted) accesses that our decision would incur but would not be made by an optimal schedule that could make random lookups only for the final top-k and would traverse index lists with different and minimal depths.

For looking up unknown scores of a candidate d in the index lists $\bar{E}(d)$, we would incur $|\bar{E}(d)|$ random accesses which are wasted if d does not qualify for the final top-k result (even after considering the additional score mass from $E(d)$). By computing the convolution histogram for $\bar{E}(d)$, we can estimate this probability as

$$P[d \notin \textit{top-k}] = 1 - p(d) \tag{8.2}$$

$$= 1 - p_S(d) \cdot q(d) \ , \tag{8.3}$$

where $p_S(d)$ is our well-known the score predictor

$$p_S(d) = P\left[\sum_{i \in \bar{E}(d)} S_i > \delta(d) \mid S_i \leq high_i\right] \tag{8.4}$$

but now using $\delta(d) = \textit{min-k} - worstscore(d) - o_j \cdot c$, and $q(d)$ is the selectivity estimator

$$q(d) = \left(1 - \prod_{i \in \bar{E}(d)} \left(1 - \max_{j \in E(d)} \frac{l_{ij}}{l_j}\right)\right) \ . \tag{8.5}$$

Note that we may incorporate all the available information about score convolutions, different index list selectivities, and correlations between tag-term pairs into the calculation of $p(d)$ in the very same way as described in Section 6.4.2. Then the random accesses to resolve the missing tag-term scores have expected wasted cost:

$$EWC_{\textit{RA-C}}(d) := |\overline{E}(d)| \cdot (1 - p_S(d) \cdot q(d)) \cdot \frac{c_R}{c_S} \ . \tag{8.6}$$

As for path conditions, the random accesses to resolve all o_j path conditions are "wasted cost", if the candidate does not make it into the final top-k, which happens if the number of satisfied conditions is not large enough to accumulate enough score mass. Recall from Section 3.4.2 that each satisfied

structural condition earns a static score mass c. The probability $q'(d)$ that d satisfies a set Y of the structural conditions is

$$q'(d) = \sum_{Y' \subseteq Y} P[Y' \; is \; satisfied] \tag{8.7}$$

$$= \sum_{Y' \subseteq Y} P[o' \; conditions \; i_1 \ldots i_{o'} \; are \; satisfied] \; , \tag{8.8}$$

where the sum ranges over all subsets Y' of Y for the o_j remaining structural conditions. $P[Y \; is \; satisfied]$ is estimated as

$$P[Y \; is \; satisfied] = \prod_{\nu \in Y} s_\nu \cdot \prod_{\nu \notin Y} (1 - s_\nu) \; , \tag{8.9}$$

assuming independence for tractability. The independence assumption can be relaxed by the covariance-based chain rule mentioned in Section 6.2.3, considering only non-overlapping subsets Y' of unresolved structural query patterns in Y. For efficiency, rather than summing up over the full amount of subsets $Y' \subseteq Y$, a lower-bound approximation can be used. For efficiency, we do not consider all subsets Y but only those that correspond to a greedy order of evaluating the structural conditions in ascending order of selectivity, thus yielding a lower bound for the true cost. Then the random accesses for path and twig conditions have expected wasted cost:

$$EWC_{RA\text{-}S}(d) := o_j \cdot (1 - p_S(d) \cdot q'(d)) \cdot \frac{c_R}{c_S} \; . \tag{8.10}$$

The next batch of b sorted accesses for all content-related index lists, incurs a fractional cost for each candidate in the priority queue, and the total cost is shared by all candidates in Q. Again, for a candidate d, the sorted accesses are wasted, if we do not learn any new information about the total score of d, that is, when we do not encounter d in any of the lists in $\bar{E}(d)$. Then the probability $q_i^{b_i}(d)$ of *not* seeing d in the ith list in the next b_i steps is defined like in Section 6.3.2 as

$$q_i^{b_i}(d) = P[d \text{ in next } b_i \text{ elements of } L_i \mid i \in E(d)] \tag{8.11}$$

$$\leq \frac{b_i}{l_i - pos_i} \cdot \max_{j \in E(d)} \frac{l_{ij}}{l_j} \; . \tag{8.12}$$

We can compute the probability $q^b(d)$ of seeing d in at least one list in the batch of size b as

$$q^b(d) = 1 - P[d \text{ not seen in any list}] \tag{8.13}$$

$$= 1 - \prod_{i \in \bar{E}(d)} \left(1 - q_i^{b_i}(d)\right) \; . \tag{8.14}$$

$$\tag{8.15}$$

Hence the probability of not seeing d in any list is $1 - q^b(d)$. In analogy to Section 6.4.2, the total costs for the next batch of b sorted accesses are shared by all candidates in Q, and this incurs expected wasted cost:

$$EWC_{SA} \quad := \quad \frac{b}{|Q|} \cdot \sum_{d \in Q} \big(1 - p_S(d) \cdot q^b(d)\big) \tag{8.16}$$

We initiate the random accesses for tag-term score lookups and for structural conditions, if and only if

$$EWC_{RA\text{-}C}(d) < EWC_{SA} \quad \wedge \quad EWC_{RA\text{-}S}(d) < EWC_{SA} \ , \tag{8.17}$$

respectively, with RAs weighted to SAs according to the cost ratio c_R/c_S. Similarly to the initial Ben-probe scheduling we always consider the *cumulated* EWCs for all batches done so far.

Note that all three EWC classes have the same basic structure that is derived from the combined score predictor and selectivity estimator described in Section 6.2.2. For each type, we count the wasted cost as individual lookups to each of the remaining index lists for a candidate depending of the type of lookup, i.e, $|E(d)|$, o_j and b, respectively. Solely the selectivity estimate $q(d)$ is exchanged based on what type of random or sorted access is about to be taken into consideration, whereas the score prediction $p_S(d)$ remains the very same for all the three types. EWC_{SA} additionally aggregates these values into an average cost over all candidates $d \in Q$.

We actually perform the random accesses one at a time in ascending order of content-related (for tag-term pairs) and structural selectivities (for navigational conditions connected to path and twig conditions). Candidates that can no longer qualify for the top-k are eliminated as early as possible and further random accesses on them are canceled. After each random access, it is tested whether the candidate document can qualify for the top-k result; if the candidate can be dismissed, all subsequent random accesses are canceled. The cost comparison and scheduling decision are made only once at the start of the entire sequence of random accesses for a candidate. Analogously to our argumentation for the content-only scheduling decisions, the additional cost comparisons for the scheduling decisions on the structure have an acceptable computational overhead and are performed whenever the priority queue is rebuilt.

8.5 Dynamic Query Expansion for Content & Structure

8.5.1 Incremental Merge & Structural Joins

Adapting the Incremental Merge method for semistructured data and queries is very straightforward and opens a variety of novel expansion options for vague search on XML data. The only obstacle compared to joining plain inverted lists is that elements and their scores are constrained through structural conditions, too; and element entries in the basic inverted index lists are grouped into element blocks in our setting, where not all elements within a block have to be assigned the same perfect score *maxscore* of that block.

In contrast to the plain text case, elements may be mutually dependent and allowing only the first expansion block per document, i.e., the one with the maximum combined similarity and element score $\max sim_{ij} \cdot maxscore$, would make us run into the danger of reporting false negatives and potentially prune candidates from the top-level queue too early. This might be the case when the structure is not matched by an element out of the first expansion block that is reported by the Incremental Merge operator for a document d, whereas lower-scored elements from expansion blocks with a lower $sim_{ij} \cdot maxscore$ value might still achieve a higher aggregated path score. However, these potential matches can easily be detected through further merging the expanded lists and iteratively polling the Incremental Merge operator for the next element block in descending order of the combined similarity and block scores.

Disjunctive Incremental Merge Operator

The only conceptual change is to skip the elimination of duplicate document occurrences in the virtual list by the Incremental Merge operator and, thus, incrementally allow more than one element block per document and query condition to be taken into the structural joins, in the sense of a *disjunctive* expansion for each virtual list. In order to still be able to guarantee a monotonous bestscore decrease for all candidates, we have to consider a more conservative $bestscore_i(d)$ bound for a candidate d at an Incremental Merge dimension i as

$$bestscore(d) \;=\; \sum_{i=1}^{m} max(worstscore_i(d), high_i) \;. \tag{8.18}$$

Note that the way the path scores are evaluated, namely through iteratively *maximizing* parent- and subtree scores at each Staircase join step, keeps the

worstscore updates monotonous, too, and exactly confirms to an Incremental-Merge-style max-score aggregation – but with regard to the structure. This practical idea elegantly closes the gap between the Incremental Merge approach and our XML evaluation strategy, now maximizing path scores instead of document-wide scores.

Since elements are read off the index in sorted order of *maxscore* values per block, the Incremental Merge now propagates whole element blocks. Note that the virtual lists for expansion terms are also organized by (*docid*, *maxscore*, *score*, *pre*, *post*), so we simply apply the block-scan technique whenever we perform the next scan step on one of these lists. Each new block that comes in by an Incremental Merge step is already sorted in descending order of element scores, and the new block can be efficiently merged into the existing element block, that a candidate already collected for a previous Incremental Merge step, in linear time without having to resort element scores at query processing time. Then further query processing is identical to the TopX algorithm for in-memory structural joins.

Incremental Merge for Tag-Term Pairs

Suppose we want to dynamically relax the combined tag-term pair `//~par = ~databas` with similar tags and terms, e.g., detected through thesaurus lookups or structural relevance feedback (see also [ST06a, ST06b] for a possible approach). Expanding multiple tag *and* term conditions in a single step gives leeway on how to combine the resulting relaxations, namely

1) *independently* through expanding the tags and terms into all $tags \times terms$ tag-term pair combinations, or

2) with *conditional dependencies*, thus expanding individual tag-term combinations, only, with the possibility to exploit explicit correlations or expansion dependencies as provided by the particular expansion technique.

The first case can easily be implemented as a *two-dimensional* Incremental Merge operator that simply consists of two nested Incremental Merge operators, one for the tag expansion and one for the term expansion. Then the resulting combined expansion similarities, and local scores for the expansions take the form $sim(tag_i, tag_{ij}) \cdot sim(t_i, t_{ij}) \cdot s_{ij}(e)$.

Note that for the second case, our default Incremental Merge strategy remains unchanged, with all expanded tag-term pairs being explicitly enumerated to initialize the Incremental Merge operator with an individual similarity score obtained through any specific expansion technique as mentioned

above. In both cases, meta histograms can be used to carry over our extended scheduling and probabilistic pruning decisions similarly to the text case, with basic input histograms being precomputed over the element full-contents.

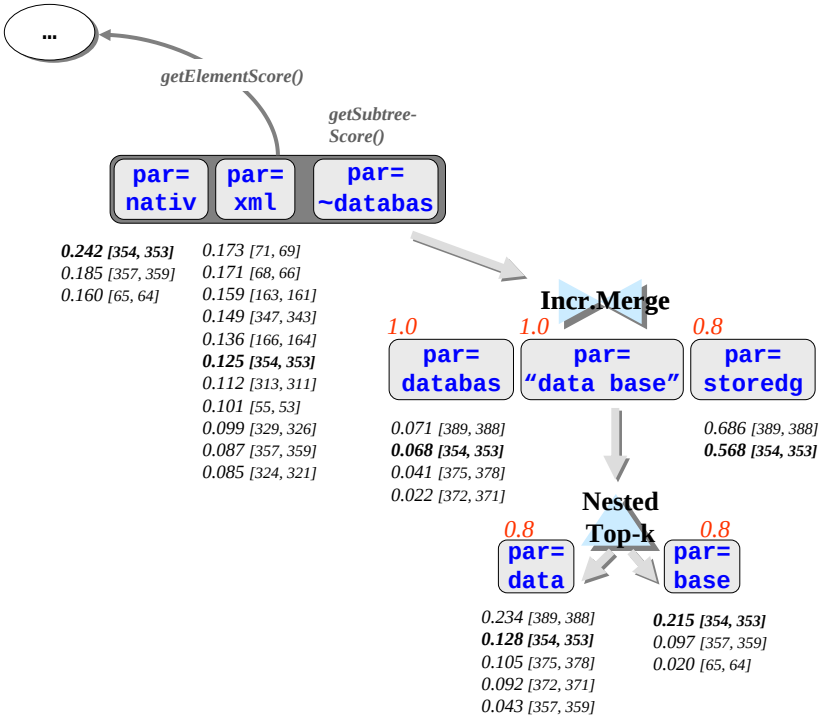


Figure 8.12: Dynamic expansion of content conditions.

Figure 8.12 depicts the situation for an example expansion of the stemmed tag-term pair `par=databas` into *exp(*`par=~databas`*)* = *{* `par=databas`, `par=''data base''`, `par=storedg`*}* with similarities 1.0, 1.0, and 0.8, respectively (we omit the tag expansion for easier readability). The Incremental Merge operator schedules the order in which element blocks from the corresponding inverted lists are merged into the element block of the candidate document for the expanded dimension. Whenever the candidate structure is changed, i.e., a new element block has been merged with the existing elements, these elements are taken into the structural joins and the worst- and bestscore bounds are updated accordingly.

Note that a nested top-k operator is utilized to generate a dynamic index list for the phrase expansion `par=''data base''` which aggregates phrase scores at the lower operator analogously to the text case – but with respect to individual element scores. Also, the lazy scheduling of phrase tests by the top-level top-k operator remains the most efficient strategy. Phrase tests

are now used to prune individual elements and do not necessarily render the whole candidate document invalid when failed for some of the elements.

8.5.2 Hybrid Index Structures

Recall from Section 2.3 that the pre/postorder labeling scheme can efficiently accelerate the descendant axis in location paths, but might degenerate for deeply nested path expressions with a low selectivity. DataGuides, on the other hand, with their ability to encode whole location paths into a single bucket id, are a perfect method to address this issue, but they do not support the descendant axis in location paths well. Although we might try to precompute all descendant path relaxations and materialize them in our inverted index for all bucketid-term pairs, this would hardly be feasible for an XML collection with a complex schema or diverse structure such as INEX.

As an example, consider the rather inconspicuous path expression

```
//article//sec//p
```

which contains three descendant-axis steps and yields exactly 520 distinct bucket ids (i.e., distinct root-to-leaf paths) in the DataGuide structure for INEX. On the other hand, the location path for

```
//movie//actor//name
```

comprises the same amount of descendant steps but yields only a single bucket id in the DataGuide for the IMDB collection which would make the DataGuide the perfect choice to process the latter path expression in IMDB.

An intriguing idea would be to perform the relaxation (for a reasonable amount of choices in the expansion possibilities) again directly in the query processor, now using our Incremental Merge approach to dynamically expand a location path with descendant steps into a number of similar paths using the child axis, only.

Incremental Merge for Bucketid-Term Pairs

Incorporating DataGuides in our structure-aware query processing requires a significant schema extension, however. Analogously to the tag-term pair content index used for the pre/postorder labeling scheme, we now index and query for bucketid-term pairs as the main building blocks for our query processing strategies. Then the new content index constitutes of the following relations:

- `DataGuide:` IOT with attributes concatenated in the order *(path, bucketid)*.

- `BuckeidTermFeaturesRA`: IOT with attributes concatenated in the order *(docid, bucketid, term, score, maxscore, pre, post)* as key.

- `BucketidTermFeaturesSA`: B^+-tree index over all attributes of `BucketidTermFeaturesRA` but concatenated in the order *(bucketid, term, maxscore, docid, score, pre, post)* as key.

- `BucketidsRA`: Separate IOT with attributes concatenated in the order *(docid, bucketid, pre, post)* as key.

Each content condition in the query now opens a sequential scan on the `BucketidTermFeaturesSA` B^+-tree index using the key prefix $(bucketid, term)$. All assumptions on random accesses for content and navigational conditions on the `BucketidsRA` index follow analogously to the pre/postorder labeling scheme (see Appendix A.1.2 for the exact schema definitions (DDL) using Oracle).

The compact DataGuide DFA (see Section 2.3.2) and all distinct path-to-bucketid mappings can typically be kept in-memory for the type of document collections we investigate. These mappings are stored in the relational schema, too, and loaded into main memory when the engine starts. For both INEX and IMDB, the memory consumption of the DataGuide is negligible; for INEX the DataGuide has about 10,000 and for IMDB only about 100 entries.

The DataGuide easily fits into main memory, the data instances of course not. Moreover, using bucketid-term pairs for querying, only provides a structural filter for element contents, since the paths do not provide a unique identifier for the elements as required for joining their scores. In particular, evaluating branching path queries only on the basis of DataGuides would make us run into the danger of returning false positives. Hence, structural joins are furthermore performed on the pre- postorder labels in the form of a *hybrid index* which basically gets us two birds in one shot:

1) We use DataGuides for query rewriting, only, and encode whole paths into a compact bucket id with lower selectivity than simple tags.

2) We perform structural joins on pre- postorder labels, and thus are able to reuse the very same join-algorithm and implementation.

The latter point even enables the query rewriter to dynamically select the most appropriate index structure for individual query nodes and to efficiently process mixed query conditions, with some navigational conditions referring to DataGuide locators and some using individual tag conditions.

Figure 8.13 depicts the approach for the example location path `//article //sec//par` that is merged into a content condition with the (stemmed) term "databas". Let us assume, the DataGuide lookup yields only three matching path with respect to the child axis, namely `/article/sec/par`, `/article/sec/ss1/par`, and `/article/sec/ss2/par`. Note that it is also possible to incorporate path similarities at this point, e.g., along the lines of [ST06a, ST06b], as indicated by the figure. An Incremental Merge operator is used to determine the order in which inverted index lists for the respective bucketid-term pairs are merged, again merging whole element blocks and propagating them for the structural joins with other element blocks at different query dimensions for each candidate.

Note that the example is a bit simplified, for the actual amount of 520 distinct bucket ids for the above path expression, we would rather keep the approved pre/postorder scheme and be willing to accept some random lookups for the navigational tag conditions `article` and `sec`, thus sequentially scanning on the tag-term pair `par=databas`, only.

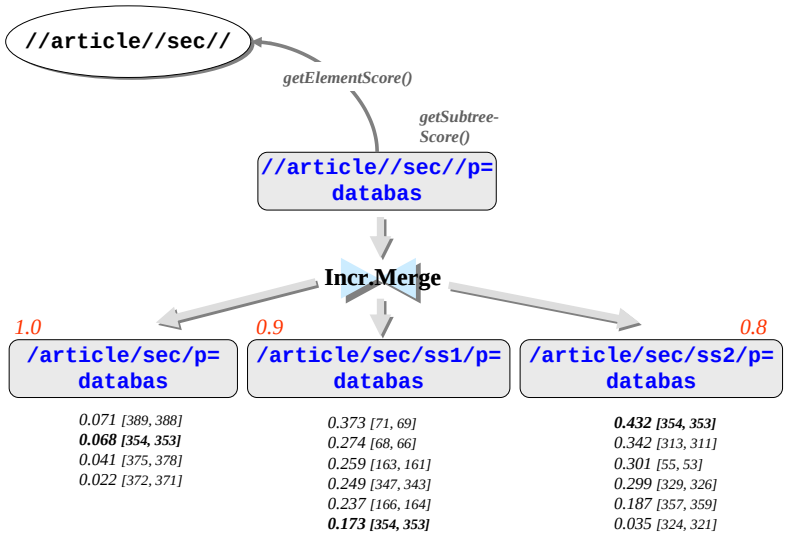


Figure 8.13: Dynamic expansion of the descendant axis for a DataGuide-like location path.

Dynamic Selection of Index Structures

The query rewriter can now incrementally query the DataGuide for all path prefixes and break the location path into a tag sequence (thus switching from DataGuides to the pre/postorder scheme) as soon as the amount of

distinct bucket ids for the path prefix exceeds a certain threshold value. The exact choice on when to keep a location path with descendant steps for being processed with a DataGuide, and when to split the path into a sequence of single navigational tags is collection-dependent. Initializing a huge amount of database cursors for the Incremental Merge algorithm may become more expensive than the actual query execution; we found a threshold of 12–24 a good choice in INEX, however. Although we do not consider DataGuides to be a general panacea for addressing a lowly selective structure, dynamically switching between DataGuides and tag-term pairs in fact allows us to efficiently cover a broad range of XML data collections with different structural characteristics.

Note that supporting DataGuides and tag-term pairs *concurrently* in our inverted block-index organization is space-consuming, since it roughly doubles the index size as scanning for bucketid-term pairs instead of tag-term pairs of course affects the grouping condition for the elements in the inverted block-index which are then grouped by $(docid, bucketid, term)$ and $(docid, bucketid)$, respectively. Keeping the two schemes in parallel would be an option for prototype tests, only. The decision on whether to index a collection using only DataGuides or only tag-term pairs depends on the amount of variations of paths in the collection, thus considering the different salient properties of each index structure (see Section 2.3). Note that the compact in-memory DataGuide DFA may be further kept for filtering invalid edges in the pre/postorder mode, too, and thus prevent the algorithm from performing unnecessary random lookups for inherently unsatisfiable structural constraints. Using the DataGuide schema is supported without any restrictions, when exclusively the child axis is used for the path queries.

Algorithm 13 In-memory top-k element extraction.

```
1:  getTopkElements(Top-k Documents documents[1..k])
2:  resultElements := ∅;
3:  threshold := documents[k].worstscore;
4:  for all i := 1..k do
5:    elements := getRankedTargetElements(documents[i]);
6:    e := 1;
7:    for all a := 1..|elements| do
8:      // We do not need more than k elements from all docs
9:      if e geq k then
10:       break;
11:     end if
12:     // No results yet, then take the first non-empty list
13:     if resultElements = ∅ then
14:       resultElements := elements;
15:       break;
16:     else
17:       b := 1;
18:       // Otherwise iterate over the result list
19:       while b < |resultElements| & elements[a].score < resultElements[b].score do
20:         resultElements.insert(b, elements[a]);
21:         // We already have k better elements
22:         if e ≥ k | resultElements[k].score ≥ threshold then
23:           break;
24:         end if
25:         b++;
26:         e++;
27:       end while
28:       // Reached end of the result list -> append all remaining elements
29:       if b = |resultElements| then
30:         resultElements.appendAll(elements[a..|elements|]);
31:         break;
32:       else
33:         // Otherwise iterate over the current element list
34:         while a < |elements| & elements[a].score > resultElements[b].score do
35:           resultElements.insert(b, elements[a]);
36:           // We already have k better elements
37:           if e ≥ k | resultElements[k].score ≥ threshold then
38:             break;
39:           end if
40:           a++;
41:           e++;
42:         end while
43:       end if
44:     end if
45:   end for
46: end for
47: return resultElements[1..k];
```

Chapter 9

Experimental Evaluation

9.1 Hardware & Software Setup

The TopX prototype search engine presented in this thesis is fully implemented in Java using J2SDK 1.5. All inverted index lists are stored as tables with appropriate B^+-tree indexes in an Oracle 10g database; in addition, a term position index is kept in a separate table for phrase matching. Auxiliary data such the thesaurus database used by the Ontology Service and index list statistics such as histograms are stored in the form of further database relations and largely cached inside the engine for multiple, interactive user sessions.

For sequential and random accesses, the search engine accesses the index lists via JDBC, with large, collection-specific prefetching buffers that are continuously filled through the background scan threads as described in Section 4.3. The query processing is multi-threaded and reads the input tuples sequentially off these buffers, with thread synchronization according to this prefetch size (e.g., after every $b = 1,000$ index scan steps).

If not stated otherwise, experiments described in this chapter were conducted on a high-end PC (Intel Dual Xeon with 3 Ghz each), 4 GB of RAM, and a large RAID-5 disk array.

9.2 TREC - Text REtrieval Conference

The annual *Text Retrieval Conference* (TREC) [TRE] provides a high level forum for state-of-the-art research on text IR. It is a unique means for standardized text collections, queries, and evaluations, with largely non-idiosyncratic relevance judgments provided by senior employees of the National Institute for Standards and Technology in Gaithersburg, Maryland, USA.

We participated in various subtasks (so-called "tracks") in 2003, 2004 and 2005 (TREC-12–14). In the first year, runs were conducted with a modified version of the BINGO! [SSTW02, STSW02] indexing framework; for the latter two years, the brand new TopX prototype came into play. We report our results for the TREC 2004 Robust, Web, and Terabyte tracks, as well as for the TREC 2005 Terabyte Efficiency task at the end of the text experiments section. Note that all TREC default query sets and relevance judgments for the various tracks and tasks are available for download at `http://trec.nist.gov/data.html`. A complete list of our extended queries for the non-standard collections and experimental setups such as IMDB and XGOV (see below) is contained in the Appendix.

9.2.1 TREC Data Collections

GOV Collection

Our GOV setting uses the data of the official GOV collection of the TREC 12–14 (2003–2005) Web Track [CHWW03, CH04] which consists of a large crawl from the `.gov` Internet domain of the U.S. government. It contains about 1.25 million documents (mostly HTML and former PDF files converted to text) and a total size of 18.1 Gigabytes. We used the original 50 queries from the TREC-12 Web Track's topic distillation task [CHWW03], which are plain keyword queries with up to 5 keywords. Examples are "legalization marijuana", "Lewis Clark expedition", "airbag injuries death".

We also participated in the Web track 2004 [CH04] which presented a mixed set of topics consisting of 75 home page finding topics, 75 named page fining topics, and 75 topic distillation queries which were not explicitly distinguished by TREC. The clue for this task was to automatically tune the system to work well for a broader range of different topic types, thus simulating typical Web search queries.

XGOV Queries

The eXpanded GOV (XGOV) setting, which we will refer to for the probabilistic pruning experiments, is not an official benchmark task. In the XGOV setting, we wanted to study the impact of the number of query-relevant index lists and *manually* modified the queries by adding synonyms and other strongly related terms to the keywords of a query. These additional terms were mostly taken from the synonym entries and descriptions of the WordNet thesaurus, where we manually identified for each original keyword the relevant word sense. This query expansion typically doubled the number

of keywords per query; the longest query contained 20 keywords (e.g., "legalization marijuana cannabis euphoric drug abuse pot smoke ...", see also Appendix A.4.3 for the whole set of expanded queries).

The GOV and XGOV settings are the basis for the evaluation of our probabilistic pruning techniques.

Aquaint Collection

The Aquaint corpus contains approximately 528,000 news articles from the LA Times, Financial Times, and the Foreign Broadcast Information Service (FBIS) in about 1.9 Gigabyte of raw text data. The TREC-13 (2004) Robust track [Vor04] explicitly identifies a distinguished set of 50 topics drawn from previous TREC ad-hoc tasks which are marked as *hard* based on result obtained by earlier submissions.

Since the focus of the Robust track is on poorly performing topics, a box plot of the average precision scores for all runs submitted to the ad-hoc task from TREC 6–8 was created to distill the most difficult queries. NIST then selected topics with low median average precision scores but with at least one (there was usually more than one) high outlier. The requirement for at least one system doing well on the topic was designed to eliminate flawed topics from the topic set.

We intensively studied this particular collection together with the 50 hard Robust queries for our query expansion experiments.

Terabyte Collection

Since GOV and Aquaint are still not exactly very large dataset in terms of documents, we included experiments on the TREC Terabyte corpus as a stress test, as well. The Terabyte collection of the new TREC-13 (2004) Terabyte track [CCS04, CCS05] is a more recent and extended crawl of the `.gov` domain and consists of more than 25 million crawled Web pages from the U.S. government domain and a total size of about 426 Gigabytes. So Terabyte is basically an extended version of the GOV collection and as such, also contains mostly HTML and PDF files converted to plain text. Again, the 50 ad-hoc-style benchmark queries are mere keyword queries such as "train station security measures" or "Aspirin cancer prevention".

We also included this benchmark in our evaluation as a stress test for the thesaurus-based query expansion with the Incremental Merge technique for inverted lists. The thesaurus-based expansion for the above two queries considered additional terms such as "railway station, railroad terminal, passenger, security system, alarm, ..." and "acetylsalicylic acid, drug, Bayer,

tablet, malignant growth, cell division, ...".

Terabyte served as a stress test for the both the scheduling and the query expansion experiments.

Terabyte Efficiency Task

In TREC-14 (2005), there was an additional Efficiency task [CCS05] introduced for the Terabyte collection with 50,000 queries from a commercial search engine. While systems had to submit results for all topics, only a small subset of them, namely the 50 manually intermixed ad-hoc topics of the same TREC, were judged by human assessors. The subset of 50 ad-hoc topics was officially announced only after the submission deadline for the Efficiency runs, however.

9.2.2 Topic Format

One particularity of the TREC queries (or so called *topics*) is that they also come shipped with larger description and narrative fields that allow for the extraction of extended keyword queries or automatic query expansion runs.

```
<top>
<num> Number: 705

<title>
Iraq foreign debt reduction

<desc> Description:
Identify any efforts, proposed or undertaken, by world governments to seek
reduction of Iraq's foreign debt.

<narr> Narrative: Documents noting this subject as a topic for
discussion (e.g. at U.N. and G7) are relevant. Money pledged for
reconstruction is irrelevant.

</top>
```

Figure 9.1: TREC ad-hoc topic format.

A typical ad-hoc run would simply extract the 2–5-term title fields for querying, whereas more elaborated expansion techniques could also take additional information from the description `<desc>` or narrative `<narr>` fields into account. Note that this may pose further challenges for an automatic system to translate these natural language descriptions into a meaningful keyword query.

9.3 INEX – INitiative for the Evaluation of XML Retrieval

The counterpart of TREC in the semistructured world is the *INitiative for the Evaluation of XML Retrieval* (INEX) [INE]. INEX queries come shipped in a similar topic format as TREC queries (but with structured CAS queries fomulated in the NEXI query language, see Section 2.2.3), but both the topic development and relevance assessments are selflessly conducted through the active participants themselves. The INEX workshop is traditionally held at the Schloss Dagstuhl International Conference and Research Center located in the south-west of Germany, very close to Saarland University and MPII. We also report results from our participation in the INEX 2005 ad-hoc retrieval task at the end of the XML experiments section.

9.3.1 INEX Collection

The *INEX* collection consists of full articles from IEEE Computer Society journals and conference proceedings in a rather complex XML format. In 2005, there was as switch form the original collection with about 12,500 documents in 500 MB raw XML data to a slightly extended version with 17,000 documents and about 750 MB data size.

We chose 46 queries from the INEX topics of 2004 for which official relevance assessments are available. These yield 22 content-and-structure (CAS) queries and another 25 content-only (CO) queries. A CO example query is `‘‘XML editors or parsers’’`, and a CAS example is `//article[.//bibl [‘‘QBIC’’] and //p[‘‘image retrieval’’]]`.

As for expectations about performance, we believe typical comparisons to Google, in particular in terms of collection sizes, are misleading. XML is much more complex than Web data, and INEX CAS queries have a significant structural part. Most experimental work on XPath, XQuery, and XML IR reported in major recent conferences used the 100 MB XMark [SWK+02] dataset (fitting into memory); some of these papers reported response times around 10 seconds for more complex XMark queries. Among the participants of the 2004 and 2003 INEX benchmarks only 7 groups reported response time figures; the best numbers were 13 seconds per query, which we outperform by a factor of 20 on similar hardware and with similar result quality. With a 64-bit processor we would expect further gains from larger memory (especially for the Terabyte setup).

In the history of INEX, most participants have focused on result quality alone; only seven groups among the 2003 and 2004 official participants have

reported response figures and the best numbers were around 13 seconds per query (on hardware comparable to ours). For the same reason, we studied query expansion for increasing query complexity even if the thesaurus-based aspects may be viewed as orthogonal to the structural side of XML.

9.4 Highly Structured Collections

9.4.1 IMDB Collection – Relational

The first IMDB setting uses the data of the Internet Movie Database (`http://www.imdb.com`) to study our methods' performance on a combination of text and structured attributes. Since we do not consider structured queries in this setting, we refer to this collection in a strictly relational sense, with disjunctive combinations of keyword queries over different attributes. The IMDB collection that we extracted from the available IMDB dump files contains about 375,000 movies and more than 1,200,000 person files (describing actors, directors, etc.), and we prepared it into a four-attribute object-relational table with the schema $Movies(Title, Genre, Actors, Description)$ where $Title$ and $Description$ refer to text attributes and $Genre$ and $Actors$ are set-valued categorical attributes. $Genre$ typically contains 2 or 3 genres, and $Actors$ were limited to those that appeared in at least 5 different movies. For similarity scores among $Genre$ values and among $Actors$, we precomputed the Dice coefficient for each pair of $Genre$ values and for each pair of $Actors$ that appeared together in at least 5 movies. So the similarity for two $Genres$ or $Actors$ x and y is set to

$$\frac{2 \cdot \#\{\text{movies containing } x \text{ and } y\}}{\#\{\text{movies containing } x\} + \#\{\text{movies containing } y\}}$$

and the corresponding index list for x then contains entries for similar values of y, too, with scores weighted with the similarity of x and y in the sense of a precomputed, static expansion.

A typical query then looks like

$$\begin{gathered} Title \subseteq \{Space\} \wedge Genre \subseteq \{SciFi\} \wedge \\ Actors \subseteq \{Harrison\ Ford\} \wedge Description \subseteq \{Robot,\ War\} \end{gathered}$$

and is evaluated in an "andish" manner. We compiled 20 queries of this kind by asking colleagues (see Appendix A.4.1 for all these queries). Note that, since we use "andish" query evaluations, our similarity scoring does not require a match to satisfy all conditions.

9.4.2 IMDB Collection – Semistructured

For a true semistructured version of the IMDB collection, we also generated an XML document for each of the movies available at the Internet Movie Database (`www.imdb.com`). Such a document contains the movie's title, plot summaries, information about people such as name, date of birth, birth place, etc. The interesting issue in using this collection is the mixture between elements with rich text contents and categorical attributes such as `Genre=Thriller` yielding many ties in local scores. Again, we asked colleagues to create 20 meaningful NEXI-style queries with structural and keyword conditions; examples are queries of the kind

```
//movie[about(.//cast//casting//role, Sheriff)]
  //casting//actor[about(.//name, Henry Fonda)]
```

thus looking for movies with Henry Fonda and an arbitrary actor in the role of a sheriff (see Appendix A.4.2 for all IMDB NEXI queries).

9.4.3 WorldCup HTTP Logs – Relational

The Internet Traffic Archive (`http://ita.ee.lbl.gov`) [ITA] provides a huge HTTP server log with about 1.3 billion HTTP requests from the 1998 FIFA soccer world championship. We aggregated the information from this log into a relational table with the schema `Log(interval,userid,bytes)`, aggregating the traffic (in bytes) for each user within one-day intervals. Queries ask for the top-k users, i.e., the k users with the highest aggregated traffic, within a subset of all intervals (like "from June 1 to June 10"); our query load consists of 20 such queries for different intervals.

9.5 Collections Summary

Note that experimental studies in the literature on XPath, XQuery, and XML IR system performance are mostly based on the XMark synthetic dataset (often using the 100 MB version which fits into memory), which is not really appropriate for our setting. We believe that INEX has become the main benchmark for XML IR. Table 9.1 shows the sizes of our test collections, Table 9.2 shows the disk resident sizes including all index structures required for sorted and random accesses as denoted in the schema definitions in Appendix A.1.1 for text and A.1.2 for XML, respectively. Index creation times were between 83 minutes for INEX and roughly 14 hours for Terabyte, using standard IR techniques such as stemming, stop word removal, and the

BM25-based scoring models described in Section 3.2.2 for text and in 3.4.1 for XML, respectively, requiring corpus-wide document or element frequency statistics of term features.

	#Docs	#Elements	#Features	Size
AQUAINT	528,155	n/a	84 M	1.9 GB
GOV	1,247,753	n/a	230 M	18.1 GB
TB	25,150,527	n/a	2,938 M	426 GB
INEX '04	12,223	12,071,272	119 M	534 MB
INEX '05	16,819	17,903,073	142 M	743 MB
IMDB	386,529	34,669,538	130 M	1,117 MB
WorldCup	n/a	n/a	1,321 M	129 GB

Table 9.1: Source data sizes of test collection used.

	INEX	IMDB	TB
Features	3.8GB	7.4GB	190.2GB
Elements	0.2GB	0.6GB	n/a
Histograms	7.4MB	11.5MB	13.5MB
Twigs&Paths	108KB	4KB	n/a
Total	**4.0GB**	**8.0GB**	**190.2GB**

Table 9.2: Index sizes for some of the test collections.

9.5.1 INEX Evaluation Strategies

The INEX benchmark pursues a wide range of different evaluation strategies and in fact offers various XML-IR-specific metrics that aim to provide a comprehensive evaluation for different tasks and topic exploration strategies. Besides the two basic querying modes, namely content-only (CO) and content-and-structure (CAS) queries, we basically distinguish three orthogonal dimensions that affect the *recall base* for the relevance judgments of XML elements that are included for the evaluation of a run. This recall base affect the maximum recall a system can achieve in terms of relevant elements:

1) Document granularity (*Fetch&Browse*) vs. element granularity,

2) for element granularity, we additionally distinguish between allowing overlapping results (*Thorough*) or not (*Focused*), and

3) the *quantization* of the recall base according to a two-dimensional scale of *specificity* and *exhaustiveness* weights for each XML element as judged by a human assessor.

A strict interpretation of the support and target element in the query, for example, reduces the size of the recall base, since only those judged elements may be kept in the recall base that match the given prerequisites.

Providing a proper methodology for relevance assessments of the various subtasks in INEX has become rather complex and in fact a research issue of its own. In the following, we merely report the most important aspects for determining the recall base of elements accounted for being relevant and for weighting (quantifying) these judgments with respect to the various INEX subtasks. For more details, please have a look at [KL05].

Document-centric vs. Element-centric Retrieval

Orthogonal to the structural demands of the query, thus distinguishing CO and CAS queries, is the granularity at which results are to be presented to the user and whether they may contain overlapping result elements or not.

- *Fetch&Browse* is a basic topic exploration mode. In this mode, all target elements are grouped per document and presented to the user in "one piece", i.e., at the same rank but with potentially different scores.

- *Thorough* tasks are evaluated using the *full recall-base* based on all the available relevance judgments. Systems will obtain a score for returning as many of the relevant reference elements as possible, including all overlapping elements.

- *Focused* tasks are evaluated using a *restricted recall-base* and based on the assumption that returned overlapping components represent only as much gain as the amount of new relevant information they contain, and that the retrieval of near-misses is considered useful to the user. That is, if a less specific or exhaustive element A that is contained in another element B, and B has been returned by the system at a higher rank than A already, A is not accounted as relevant for the submission.

Further subtasks in the basic element modes may further restrain the recall base depending on whether the support elements are interpreted as vague and the target element as strict (SVCAS), or whether the support elements are strict and the target element is vague (VSCAS), or, finally, both types are vague (VVCAS), or both types are strict (SSCAS).

Note that controlling overlap is inherently difficult for a top-k engine to be performed at query processing time for the two subtasks that specify vague target elements, namely VSCAS and VVCAS, because they accept different element types as valid results, e.g., potentially large section that contain smaller embedded paragraphs elements which are also returned as separate results. That is, overlap removal as demanded by the *Focused* tasks is typically a result of a post-processing step; see also [Cla05] for a nice overview of these strategies.

For the SSCAS or Thorough tasks, the problem of overlapping results does not arise (following a strict evaluation strategy). Thus, we restrict our INEX experiments (see Section 9.8.4) to either the SSCAS or Thorough evaluation strategies.

Quantization

The quantization function f defines the weight at which relevant elements are accounted for the recall base. Unlike the binary relevance judgments provided in TREC, i.e., "Not Relevant" and "Relevant", relevance judgments in INEX are distinguished at a two-dimensional scale of *specificity* and *exhaustiveness* judgments. These range from "Unspecific" to "Highly Specific" and "Not Exhaustive" to "Highly Exhaustive" in a four-point scale each.

- *Specificity* is determined as an estimate of the fraction of the amount of relevant information about the topic-of-interest versus the amount of overall information (including other topics) that is captured by the result element.

- *Exhaustiveness*, on the other hand, is determined as the fraction of the amount of relevant information that is captured by the result element versus the amount of overall available information about the topic-of-interest.

The relevance degree of an assessed component c, given by the combined values of exhaustiveness and specificity, is then denoted by pairs $(e, s) \in ES$, where $ES = \{(0,0), (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$, and referred to as $assessment(c)$. Near misses, however, that may also guide the user the a relevant piece of information can be taken into account for the (e, s) evaluation. The two most important quantization functions are the *strict* and *generalized* quantizations which are defined as follows:

- The *strict quantization* only accounts result elements that have been

judged with the maximum value of $(e, s) = (3, 3)$.

$$f_{strict}(e, s) = \begin{cases} 1 & \text{if } 3/3 \\ 0 & \text{otherwise} \end{cases} \tag{9.1}$$

- The *generalized quantization* also accounts those elements that do not have the maximum (e, s) judgment. Individual exhaustiveness and specificity pairs are mapped onto discrete weights for determining the recall.

$$f_{generalized}(e, s) = \begin{cases} 1 & \text{if } 3/3 \\ 0.75 & \text{if } 2/3,\ 3/2,\ \text{or } 3/1 \\ 0.5 & \text{if } 1/3,\ 2/2,\ \text{or } 2/1 \\ 0.25 & \text{if } 1/1,\ \text{or } 1/2 \\ 0 & \text{if } 0/0 \end{cases} \tag{9.2}$$

Then the number n of relevant components in the recall base with respect to a given topic is calculated as

$$n = \sum_{c \in components} f_{strict|generalized}(assessment(c)) \tag{9.3}$$

which is taken into account to derive the default *precision/recall* curves and MAP values [BV00, Vor04] for the final system rankings.

Note that the new INEX 2005 metrics [KL05] also foresee a new quantization function $f_{quant} : ES \rightarrow [0, 1]^3$ with continuous scores that are automatically derived from an interactive topic assessment tool that allows for the manual selection of individual text passages for each result element.

9.6 Evaluation Metrics

We employed various metrics to evaluate our approaches with respect to the given benchmark task. Note that we cannot report results for all possible combinations of tasks and metrics at this point, we rather try to apply a choice of the most appropriate metrics with respect to the given query task, mostly being derived from the official TREC or INEX benchmark guidelines for transparency.

For efficiency comparison, we collected the following measures:

- *#SA* – the total amount of sorted accesses to inverted index lists for all benchmark queries, thus counting the number of individual tuples (or index list rows) that are read sequentially

- *#RA* – the total amount of random accesses to the index lists and the position index (for phrase matching), thus counting the number of individual tuples (or index list rows) that are read through random access for some object id
- *COST* – the total cost of executing all queries denoted as $\#SA + c_R/c_S \cdot \#RA$ according to the cost ratio c_R/c_S
- *#CPU* – the total CPU time for all benchmark queries in CPU seconds altogether (e.g., for a whole batch of 50 queries)
- *#sec* – the total wallclock runtime for all benchmark queries in seconds altogether (e.g., for a whole batch of 50 queries)
- *KB* – the maximum memory consumption in Kilobytes during the benchmark run (for maintaining the candidate pool priority queue, etc.)
- q – the maximum queue size q created with regard the different queuing strategies in the probabilistic pruning setup (as opposed to the mere KB value, this yields an intuition for the absolute number of candidates kept in the queue)

For assessing the quality of the approximate top-k query results, we collected the following measures:

- *precision* – the fraction of top-k results in an approximate result that belongs to the true top-k result (this is equal to the definition of *recall* when comparing the exact top-k result set with the approximate top-k results obtained from probabilistic pruning with $\varepsilon > 0$), i.e.,

$$\frac{\text{true-top-}k \cap \text{approx-top-}k}{k}$$

- *rank distance* – the footrule distance [KG90] between the ranks of the approximate top-k results versus their true ranks in the exact top-k result, i.e.,

$$\frac{1}{k} \sum_{i=1}^{k} |\text{true-rank}(i) - i|$$

- *score error* – the absolute error for approximate versus exact top-k scores, i.e.,

$$\frac{1}{k} \sum_{i=1}^{k} |\text{true-score}(i) - \text{approx-score}(i)|$$

- *P@k* – the macro-averaged absolute precision for the top-k results of each query, using official relevance assessments provided by TREC or INEX
- *MAP* – the non-interpolated mean average precision (MAP) [BV00, Vor04] displays the absolute (i.e., user-perceived) precision as a function of the absolute recall, using official relevance assessments provided by TREC or INEX

The following TREC-specific metrics were used by the Web track [CH04] for the *home page* and *named page* finding tasks, since these aim at only a single target page:

- *MRR* – the mean reciprocal rank (MRR) is the macro-average over all queries of the reciprocal of the ranks of the target pages in the result lists
- *S@k* – the success at the top-k results reflects the absolute number of queries for which the target page has been found among the top-k results

The following XML-specific metrics were newly introduced for the INEX benchmark 2005 [KL05]:

- *nxCG* – the normalized extended Cumulated Gain metrics is an extension of the cumulated gain (CG) metrics which aims to consider the dependency of XML elements (e.g., overlap and near-misses) within the evaluation, and
- *ep/gr* – the precision-precision/gain-recall metrics finally aims to display the amount of relative effort (where effort is measured in terms of the number of visited ranks) that the user is required to spend when scanning a system's result ranking compared to the effort an ideal ranking would take in order to reach a given level of gain relative to the total gain that can be obtained

All averages are reported as *macro-averages* over the whole set of benchmark queries to smooth the otherwise high variance induced by few extraordinarily short- or long-running queries. Note that we did not use default rank-comparison measures such as Spearman's rank correlation or Kendall's τ [KG90] as we wanted to assess only the top-k ranks of the approximate result rather than all ranks, but the Spearman and Kendall measures require comparing two permutations of the same sets of possible ranks (assuming 100 percent overlap). Further note that the relative precision and

recall have identical values in our setup, because both metrics use the same denominator k. The baseline for precision and recall is the top-k result of the exact top-k baseline algorithm (e.g., Fagin's orginal TA, NRA, or CA algorithms [Fag02, FLN03] with respect to the given task) or the non-approximative `Prob-con` and `Prob-prog` setups with $\varepsilon = 0$.

9.7 Text IR

9.7.1 Probabilistic Candidate Pruning

We will first turn our attention on the evaluation of the probabilistic candidate pruning component for plain inverted lists. The case for XML is described later.

Scoring Models

In order to evaluate the behavior of different parameterized score predictors and our histogram approach, we employ different scoring functions ranging from a typical IR scoring model to synthetic (clean) distributions, that are representing different applications and lead to a radically different pruning behavior for a top-k algorithm, even when no probabilistic pruning is used.

1) the original scores computed using TF·IDF products, with TF and IDF normalized by the maximum TF value per document and the strongly dampened IDF value as described in Section 3.1.1,

2) randomly assigned scores with a $(0, 1]$ Uniform distribution,

3) randomly assigned scores with a Zipf distribution starting from low scores, so that low scores are much more frequent

For both the synthetic Zipf and Uniform distributions, merely the scores among list entries were exchanged, while the object ids in the index list entires were not changed, i.e., the ordering of items within each list and the overlap across lists (correlations) were preserved.

Queuing Strategies

The algorithms compared in the experiments are the four Prob-k methods presented in Section 5.3:

- `Prob-con`: the conservative algorithm,

- `Prob-agg`: the aggressive algorithm,
- `Prob-pro`: the progressive algorithm, and
- `Prob-smart`: the smart algorithm.

Competitors

For each of them, we considered different options for probabilistic prediction. The baseline against which we compare our methods is:

- `NRA`: the original No-Random-Access algorithm with sorted access only, using the implementation as discussed in Section 5.3.4.

All algorithms access index lists in the baseline round-robin manner and cache large index blocks in memory. In order to be able to directly compare the effectiveness of the probabilistic pruning component, no random accesses were allowed for this setup which makes the NRA algorithm the most natural candidate for being employed as competitor in this setup. In the following, we explore a variety of scoring and predictor combinations as well as different queuing strategies for the GOV, XGOV, and IMDB (in the relational version) settings. Since no random accesses are used, #SA is our primary cost measure.

Baseline Pruning Runs

In the baseline experiment, all probabilistic predictors use histograms with cell-width 0.01 (i.e., $n = 100$ bins for each basic histogram). Convolution histograms were precomputed at query initiation time, and the impact of changing $high_i$ values was taken into consideration by periodically (i.e., every $b = 200$ sorted-access steps) rebuilding the remaining parts of the convolution histograms (see Section 5.2.3). We set the probabilistic prediction confidence level to 90 percent, i.e., ε is set to 0.1. For the smart strategy with a bounded priority queue the queue size was set to $b = 200$ entries. All access costs and runtime values were measured for the 50 GOV benchmark queries with $k = 20$.

Table 9.3 shows the performance results for the five algorithms under comparison for the GOV, XGOV, and the IMDB settings, respectively. For GOV, the chart is based on the original, TF·IDF-derived scores. We present the three efficiency metrics in terms of benchmark totals over all queries, and the three result quality metrics as macro-averages over all queries. For the GOV setting, micro-averaged values are heavily biased by a few long-running queries; for these queries the performance gains of Prob-k over NRA are even

	#SA	#sec	q	KB	prec	rank dist.	score err.
	GOV						
NRA	2,263,652	148.7	10,849	87	1.00	0.00	0.00
Prob-con	993,414	25.6	29,207	235	0.87	16.9	0.01
Prob-agg	20,435	0.6	0	52	0.42	75.1	0.09
Prob-pro	1,659,706	44.2	6,551	52	0.87	16.8	0.01
Prob-smart	527,980	15.9	400	52	0.69	39.5	0.03
	XGOV						
NRA	22,403,490	7,908	70,896	571	1.00	0.00	0.00
Prob-con	10,165,677	6,448	51,893	418	0.90	10.9	0.04
Prob-agg	133,745	2	0	101	0.35	80.7	0.18
Prob-pro	20,006,283	1,791	12,435	101	0.95	9.3	0.03
Prob-smart	18,287,636	1,066	400	101	0.88	14.5	0.04
	IMDB (Relational)						
NRA	1,003,650	201.9	12,628	103	1.00	0.00	0.00
Prob-con	463562	17.8	14,990	121	0.71	119.9	0.18
Prob-agg	41,821	0.7	0	74	0.18	171.5	0.39
Prob-pro	490,041	69.0	9,173	74	0.75	122.5	0.14
Prob-smart	403,981	12.7	400	74	0.54	126.7	0.25

Table 9.3: Baseline runs comparing NRA and the Prob-k family of algorithms (at $\varepsilon = 0.1$) for GOV, XGOV, and IMDB.

significantly higher than the macro-averaged values indicate. The XGOV results show significantly increased access rates (#SA) and queue sizes – a true stress test to the queue management, in particular for `Prob-con`.

Efficiency

The results demonstrate the significant cost savings that the Prob-k family of algorithms can achieve compared to NRA. In terms of the number of sorted accesses the conservative algorithm `Prob-con` gains more than a factor of two, and the smart algorithm `Prob-smart` achieves even a factor of four for GOV. In terms of runtimes, the two probabilistic algorithms even reduce the cost by an order of magnitude. Figure 9.2 shows that the wallclock elapsed time drops much faster than linear with increasing ε. Note that the runtime is not simply a linear function of the sorted accesses but reflects also cache and queue management overhead that is drastically reduced by the probabilistic candidate pruning. Also recall that the numbers in Figure 9.2 are total runtime, for all benchmark queries together.

For the GOV and IMDB settings, `Prob-con` temporarily even created a larger queue than the NRA baseline using periodic garbage collection, but

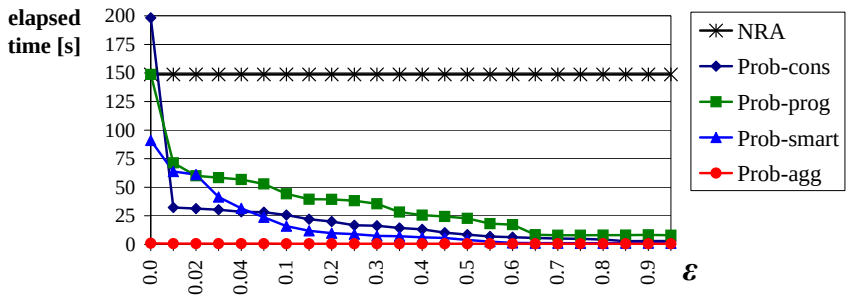


Figure 9.2: Wallclock elapsed runtime for GOV as a function of ε, for $k = 20$.

`Prob-con` dropped this large queue quickly after initialization and then distributed the remaining candidate items to multiple small queues. Interestingly, the progressive algorithm `Prob-pro` did not do as well as expected; its capabilities for early pruning are limited and its queue management, which required many insert and delete operations, is a significant cost factor. The aggressive method `Prob-agg` outperformed all competitors, but as expected, its result quality was rather poor; so we would not really consider it a winner.

For individual queries, especially those that involve very long index lists, the savings are even more impressive. For example, for the GOV query "weather hazard extremes" `Prob-con` and `Prob-smart` needed 25,802 and 19,401 sorted accesses with runtimes 0.59 and 0.68 seconds, whereas NRA required 160,002 accesses and ran in 55.60 seconds (for the 50 queries with $k = 20$, at precision 0.9 and 0.75, resp.). In the IMDB setting, the reductions of sorted accesses were not as high as for GOV but the runtime reductions reached a factor of about 10 at a high macro-averaged precision of 0.71 to 0.75. A typical query like "*Genre* $\subseteq$ *{Western}* $\wedge$ *Actor* $\subseteq$ *{John Wayne, Katherine Hepburn}* $\wedge$ *Description* $\subseteq$ *{Sheriff, Marshall}*" required 10,802 sorted accesses for both `Prob-con` and `Prob-smart` with runtimes 0.41 and 0.51 seconds, whereas NRA performed 26,402 accesses in time 4.92 seconds (for k=20, both at precision 0.7).

Finally, for XGOV queries with more keywords per query, the overhead for queue management and probabilistic predictions became a truly decisive issue. Among the methods with acceptable to very good precision, `Prob-con` performed best in terms of sorted-access savings, but the runtime gains were only modest because of the overhead of maintaining up to $2^m - 1$ queues. `Prob-pro` and `Prob-smart` were the clear winners in terms of runtime, with acceleration factors up to 8 compared to NRA. Interestingly, these methods did not save that many sorted accesses but benefited greatly from their efficient queue management.

Result Quality

Figure 9.3 depicts that the metrics for result quality show very good values for `Prob-con` and `Prob-pro` and still acceptable results for `Prob-smart`. `Prob-con` and `Prob-pro` achieved nearly 90 percent precision (and the same recall) for the GOV setting. For the IMDB setting, the precision figures were worse, one reason being that Genre scores had a major influence on the overall ranking and the small number of different values led to a fairly discontinuous score distribution with big gaps and many ties, causing some inaccuracy of probabilistic predictions. We will discuss the influence of the various predictors thereafter.

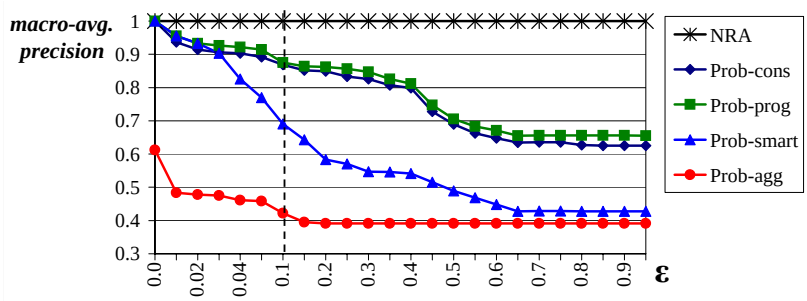


Figure 9.3: Precision for GOV as a function of ε, for $k = 20$.

The other two result-quality metrics show that the user-perceived "loss" of an approximate result actually seems well tolerable. The average rank distance for `Prob-con` and `Prob-pro` was only around 16. For $k = 20$ or higher this seems acceptable, in particular, when we consider that the average rank distance is dominated by a few outliers with very high rank distance. In terms of score error, the loss even seems negligible. By and large, the objects that are returned by the Prob-k algorithms are nearly as good as the exact top-k results.

Again, the results for the IMDB setting were not quite as good as for GOV. We manually inspected a fair number of the results and found that in most cases the results would be considered as good matches by a human user. For example, the query "*Genre*$\subseteq${*Thriller*} $\wedge$ *Actor*$\subseteq${*Arnold Schwarzenegger*} $\wedge$ *Description*$\subseteq${*robot*}" returned top results Terminator3, The 6th Day, Total Recall, Die Hard 2, Star Wars IV, etc. (recall that top-k results do not necessarily have to satisfy all query conditions).

For XGOV, `Prob-con`, `Pro-pro`, and `Prob-smart` showed very good precision, rank distance, and score error values.

Parameter Sensitivity

We studied the influence of several parameters on the performance of our four Prob-k algorithms: the probabilistic prediction confidence level 0.1, the result size k for top-k queries, the number n of bins per basic histogram, and the maximum size b of a bounded priority queue for the smart algorithm. Figure 9.4 shows the results for varying the parameter ε (the vertical dashed line is the baseline setting). The curves show that for ε below 5 percent, the progressive and smart algorithms achieve only marginal savings. For ε between 5 and 20 percent, on the other hand, these two methods offer excellent benefit/cost ratios. The conservative `Prob-cons` method performs best according to the theory of probabilistic guarantees. Already small ε values like 1 percent lead to significant cost savings, and even for ε values as large as 50 percent, which results in sorted-access savings of more than a factor of 4, still yield 70 percent precision compared to the non-approximative top-k results. The aggressive method `Prob-aggr` always exhibits great cost savings, but this is at the expense of precision values of 40 to 50 percent only. Still, this may possibly be the preferred method in applications with tight response time demands.

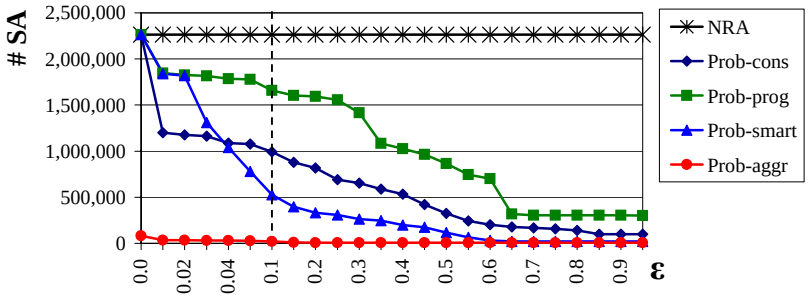


Figure 9.4: Efficiency (in #SA) for GOV as a function of ε, for $k = 20$.

We also compared the measured precision with the expected precision that we predict as a function of ε according to the formulas of Section 5.1. For `Prob-con` and `Prob-pro`, the prediction model is fairly accurate. The absolute difference between predicted and measured precision is only one or two percent for ε values of 0.2 or less; it increases for larger ε, but this prediction is conservative in that it lower-bounds the measured precision.

Figures 9.5 and 9.6 show that, with increasing k which was varied between 1 and 200, all methods exhibit linearly increasing sorted-access costs but with different gradients. For large k, the gains of `Prob-con`, `Prob-pro`, and `Prob-smart` compared to NRA are even higher than in the baseline setting; at the same time the precision of the approximate top-k results becomes even

better for high k.

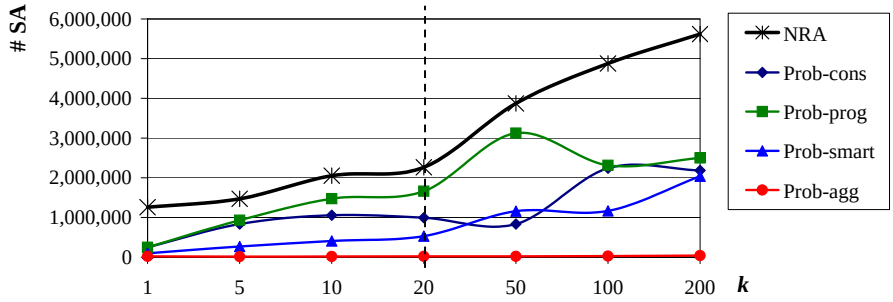


Figure 9.5: Efficiency (in #SA) for GOV as a function of the number k of returned top-k results, with $\varepsilon = 0.1$ fixed for the probabilistic algorithms.

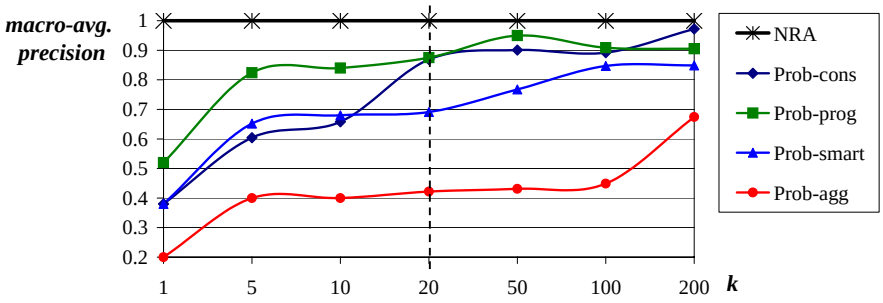


Figure 9.6: Precision for GOV as a function of the number k of returned top-k results, with $\varepsilon = 0.1$ fixed for the probabilistic algorithms.

Figures 9.7 and 9.8 show that the performance for different values n of histogram bins is fairly stable over a wide range of settings. Between 50 and 1,000 bins the relative performance of the different algorithms does not change much; below 50 bins the `Prob-smart` method does not work that well anymore, but `Prob-con` and `Prob-prog` remain more robust and show consistently good performance even with down to 25 bins per histogram. Similarly, we found that the maximum queue size parameter b for the `Prob-smart` algorithm is largely uncritical. Queue size limits as low as $b = 100$ with our iterative pruning technique (see Section 5.3) still worked very well for $k = 20$.

Predictor Sensitivity

Finally, we compared our different approaches for probabilistic prediction: histograms vs. Poisson approximations vs. Chernoff bounds based on the assumption of Uniform distributions vs. Chernoff bounds considering term

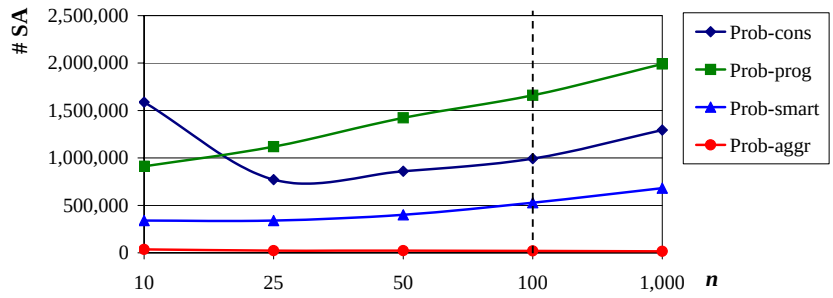


Figure 9.7: Efficiency (in #SA) for GOV as a function of the number of histogram buckets n, with $\varepsilon = 0.1$ fixed for the probabilistic algorithms.

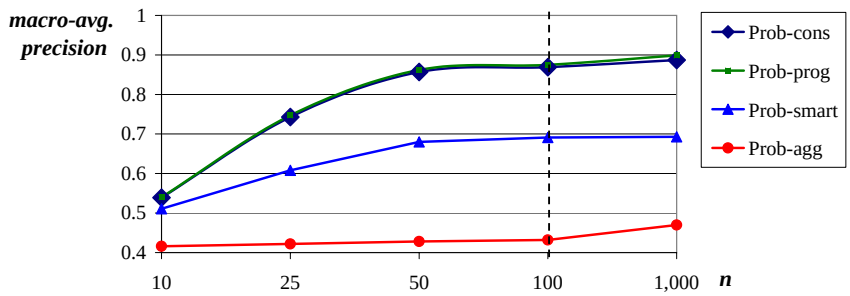


Figure 9.8: Precision for GOV as a function of the number of histogram buckets n, with $\varepsilon = 0.1$ fixed for the probabilistic algorithms.

correlations. We limit the presentation to results for the `Prob-con` algorithm with $k = 20$, $\varepsilon = 0.1$, $n = 100$, and the GOV setting. Due to the high overhead of computing Chernoff bounds with OpenMaple, we removed the three most expensive queries from the Web tracks's topic distillation task which consumed about 40 percent of the overall runtime with 50 queries.

Figure 9.9 shows the performance comparisons for the original TF·IDF scores. The dashed line is the predicted precision (for `Prob-con` this is simply a linear decrease at $1 - \varepsilon$). Similar experiments have been run for two artificially generated Uniform- and Zipf-distributed scores on the GOV index lists as shown in Figures 9.10 and 9.11, respectively.

The charts show that histograms generally provide the most accurate score predictions. A comparison with the Uniform- and Zipf-distributed scores shows that they are a flexible solution to capture different score distributions already for n as low as 50 to 100 buckets in a score domain of $(0, 1]$. Both the Poisson estimator and, particularly, the Chernoff-bound method (the latter assuming Uniform-distributed scores) are overly conservative and overestimate score probabilities for the TF·IDF and the Zipf case. The differ-

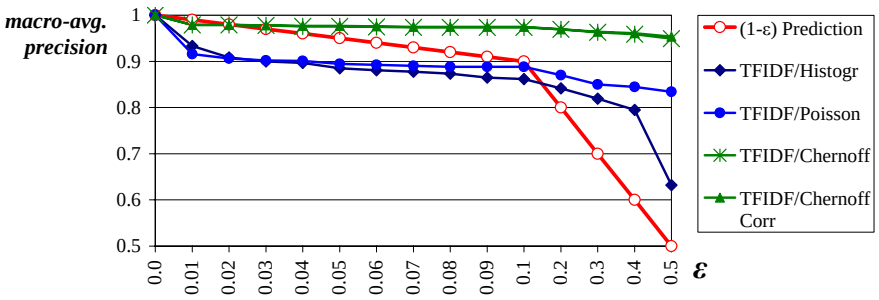


Figure 9.9: Precision of various predictors for TF·IDF-distributed scores for GOV as functions of ε, for $k = 20$.

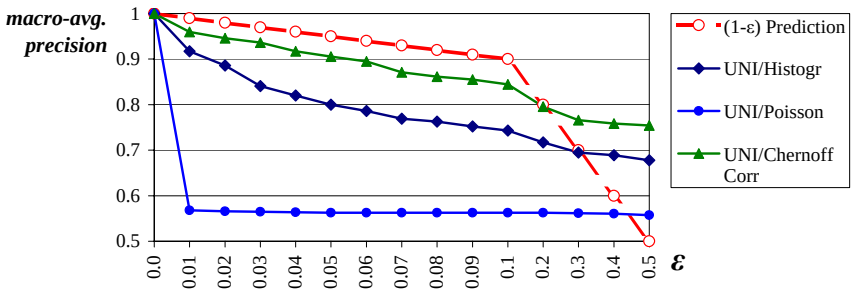


Figure 9.10: Precision of various predictors for Uniform-distributed scores for GOV as functions of ε, for $k = 20$.

ence between the Chernoff-bound methods with and without independence assumption is not really significant for the GOV data, but this could be different in other settings. For Uniform-distributed scores the Chernoff-bounds are fairly accurate over a wide range of ε, whereas, as expected, Poisson estimators do not work well for the Uniform case, because they substantially underestimate the tail probability. The Zipf distribution is closer to the original TF·IDF score distribution, but has a longer tail of low scores, for which the Poisson estimator works better, but, again, the Chernoff-bounds behave overly conservative.

The advantage of the Poisson approximation method is its very little overhead, whereas the overhead of the histogram method increases with dimensionality. But note that even with the higher-dimensional XGOV workload, the algorithms still achieved major runtime gains using histograms with dynamic convolutions. The Chernoff-bound predictors are largely independent of the dimensionality, but they suffer from huge startup costs for invoking OpenMaple. For a practically viable solution one would have to hand-code the OpenMaple computations (which involve differentiation and finding roots

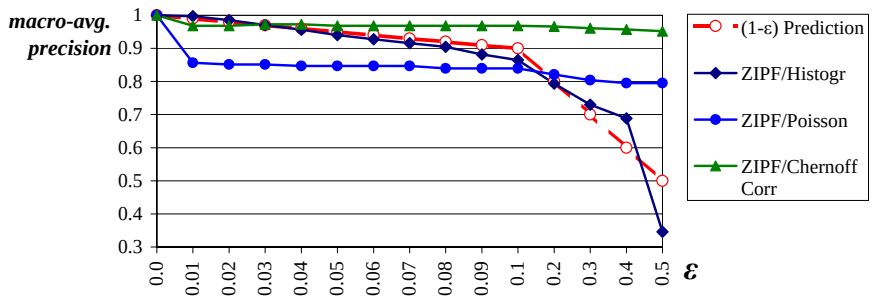


Figure 9.11: Precision of various predictors for Zipf- distributed scores for GOV as functions of ε, for $k = 20$.

numerically) in C or C++.

Finally, our model for precision guarantees developed in Section 5.4 works very well for the `Prob-con` algorithm. The `Prob-pro` and `Prob-smart` algorithms deviate from this basic statistical model, because they merge all candidates into a single queue and `Prob-smart` even bounds this queue and heuristically stops after testing the top item, only. Here the predictions were reasonably accurate for small values of ε but degraded and became overly conservative for ε higher than 5 percent, as the pruning behavior of the heuristic extensions progressively increases.

Discussion of Probabilistic Pruning Experiments

Our comprehensive experiments on various collection show that the probabilistically enhanced algorithms can achieve major performance gains, in terms of both sorted accesses and actual runtime, and at the same time provide probabilistic guarantees for result precision and recall. Among the four competing algorithms, `Prob-con` and `Prob-smart` turned out to be the most interesting ones. `Prob-con` is closest to the theory of probabilistic guarantees and does best in terms of result quality; `Prob-smart` offers the *best benefit/cost ratio*. Both methods achieve runtime gains by an order of magnitude compared to NRA. All four Prob-k methods are fairly robust with regard to parameter settings; there is no need for sophisticated tuning. For score predictions, we believe that histograms are the best choice from an engineering viewpoint, but the other two methods showed good results and certainly deserve further studies, too.

9.7.2 Index Access Scheduling

We consider three structurally different data collections: the TREC Terabyte collection, movie data from IMDB, and the huge WorldCup HTTP server log. As competitors for our algorithms we chose the most important variants of threshold algorithms, namely TA, NRA, and CA, and a baseline Full Merge that computes the full join of all index lists using a merge join, partially sorts the results by score, and outputs the top-k results. Additionally, we empirically compute a lower bound for the cost of any threshold algorithm (see Appendix A.3.2 for details) to assess how close our algorithms get to the optimum.

We also ran our experiments for the RA-extensive threshold algorithms `TA`, `Upper` [BGM02, MBG04] and `Pick` [BGM02] (see Section 1.4). In our setting, where both sorted and random access is possible and a random access is much more expensive than a sorted access (the lowest ratio we consider is 100), all these methods performed considerably worse than even the Full Merge baseline, in terms of both costs and running times, and for all values of k and c_R/c_S we considered. For example, for k=10 and c_R/c_S=1,000 on Terabyte-BM25, they resulted in total cost 72,389,140 (`TA`), 31,496,440 (`Upper`), and 3,798,549 (`Pick`), compared to 2,890,768 for the Full Merge, 788,511 for `NRA` and 386,847 for our best method. We therefore did not include these methods in our charts. Note that, as we discussed in Section 1.4, `MPro`, `Upper` and `Pick` were actually designed for a different setting, where some lists are accessible by random access only.

We focus on experiments with the Terabyte collection using a BM25 (see Section 3.1.1) and a TF·IDF model (see Section 3.2.2), both with scores normalized to $(0, 1]$. Our main attention is turned on the BM25 model, however, as this is the most challenging and most realistic IR scoring model; main results for the two other collections are presented afterwards. For further sensitivity studies we also employed two synthetically generated Zipf and Uniform score distributions for this collection.

We used a default cost ratio $c_R/c_S = 1,000$; and the inverted lists are logically divided into large blocks of size $b = 32,768$, i.e., the batch sizes b_i determined by the sorted access scheduler are multiples of this value to accommodate the size of the collection and to be able to solve the NP-hard Knapsack SA scheduling tasks *exactly*. We report the average query cost as $COST = \#SA + c_R/c_S \cdot \#RA$ computed over the whole batch of queries as our primary performance measure.

Modified Prototype

For reporting the wallclock runtime figures for the scheduling experiments, we conducted a new prototype system implemented in C++ and using raw disk access to inverted files, with large blocks of $(docid, score)$-tuples efficiently being merge-joined in-memory. The average runtime gains we achieve with this highly specialized implementation compared to the default TopX engine implemented in Java and reading tuples from Oracle via JDBC are at an amazing factor of up to 20 with no loss in precision. Note that this factor holds in particular for very large collections such as Terabyte, because we need to fetch the first block for each query term (which roughly confirms to a disk sector, e.g., using a block size of 32,000 tuples) also for a short list completely into memory which diminishes the runtime advantage of the block-merge for small collections, where the average index lists size does not even exceed a disk sector (see [BMS+06] for a full description of the implementation). The abstract cost measures based on index accesses, however, remain directly comparable to TopX.

The runtime for the scheduling experiments was measured using a two-processor Opteron 250 server with 8 Gigabytes of memory and all data loaded from a large SCSI RAID. With this new prototype, we would rank at the very top of real wallclock runtime figures reported for Terabyte so far (see Figure 9.44) for a non-distributed engine. This direction certainly deserves further attention in our future work.

Baseline Scheduling Runs

Figure 9.12 presents the average cost savings of our best approach (`KSR-Last-Ben`) for Terabyte which outperforms all our three baselines by factors of up to 3. Even for $k = 1,000$, there is a 50 percent improvement over all three baselines. Note that the end-user consumption of top-k results (as in Web search) would typically set k to 10–100, whereas application classes with automated result post-processing (such as multimedia retrieval) may want to choose k values between 100 and 1,000. Especially remarkable is the fact that we consistently approach the absolute lower bound by about 20 percent even for large k, whereas both CA and NRA increasingly degenerate; CA even exceeds the Full Merge baseline in terms of access cost for $k > 200$. Note that we measured the average query cost for the original TA algorithm with full random lookups for each candidate (`RR-All`) with a value of 72,389,140 which could not even be plotted on the same scale as the other variants for the default cost ratio $c_R/c_S = 1,000$.

Figure 9.13 shows that the average runtimes we achieve per query are

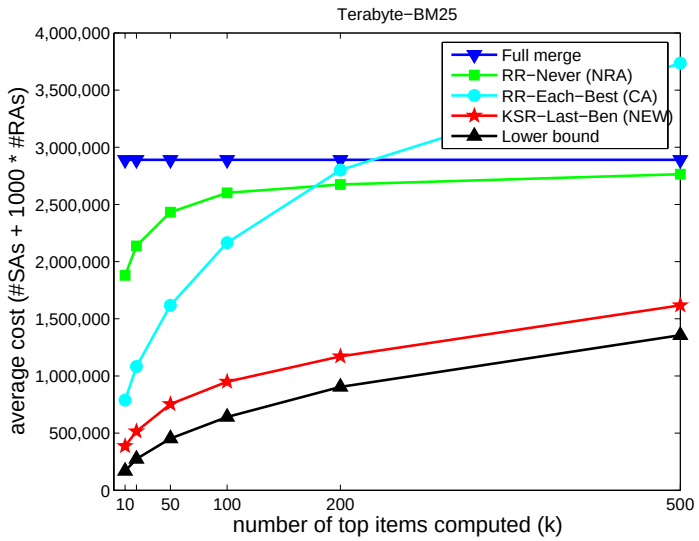


Figure 9.12: Average costs for Terabyte-BM25 of our best algorithm compared to various baselines and a computed lower bound, for varying k.

in the order of 30–60 milliseconds for $10 \leq k \leq 100$, even when the total list length is in the millions, which outperforms the NRA and Full Merge baselines by a factor of up to 5. Interestingly, for $k > 20$ our true baseline for measuring runtimes is no longer CA, because it is already outperformed by the DBMS-style Full Merge. Here, NRA is already out of the question because of its high overhead in index access costs (Figure 9.12) and its additional need for candidate bookkeeping, whereas the amount of access costs saved by our improved scheduling approaches (`KSR-Last-Ben`) more than compensates the bookkeeping overhead. To pick just one example, the Full Merge on the query "Kyrgyzstan United States relations", which has a total list volume of over 15 million document ids, takes about one second, while our best top-k algorithms, by scanning only about 2 percent of this volume and by doing about 300 well-targeted random lookups, process the same query in about 10 milliseconds.

Sorted Access Scheduling

To analyze the benefit of our Knapsack-driven SA scheduling approaches, we fix the RA scheduling to the `Last-Best` strategy and focus on the individual SA scheduling performance of the two Knapsack optimizations `KSR` and `KBA`. Note that the following figures report our results with the *exact* Knapsack problem being solved at query runtime. No greedy approximations were required for the Terabyte setting using large sorted access batching blocks

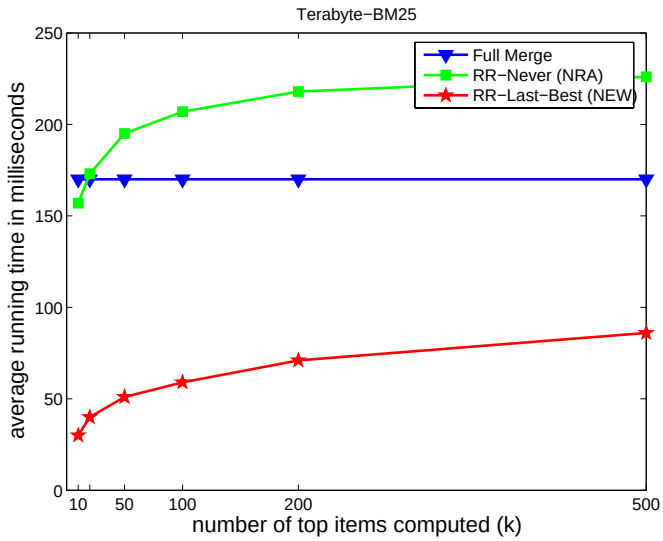


Figure 9.13: Average running times in milliseconds of our best algorithm compared to Full Merge and NRA, for Terabyte-BM25 and varying k.

of multiples of size $b = 32,768$ and the 4–5 keyword queries of the Terabyte track.

Figure 9.14 shows relatively low performance gains in between 2–5 percent for BM25 scores compared to round-robin. For more skewed distributions such as TF·IDF, we observe larger benefits of up to 15 percent for $k \geq 50$. Here, the more sophisticated benefit-optimized Knapsack (`KBA`) wins overall (see Figure 9.14).

Random Access Scheduling

Now we fix the SA scheduling to the basic round-robin (`RR`) strategy and analyze our different RA scheduling approaches. Figure 9.16 shows that we gradually improve our RA scheduling performance as we move from the original CA baseline over the simple `Last-Best` strategy toward the more sophisticated cost-driven scheduling `Last-Ben`. Interestingly, the step from `RR-Each-Best` (CA) to `RR-Last-Best` already provides 90 percent of the overall gain we can achieve, whereas the more complex `RR-Last-Ben` achieves about 10 percent more cost savings with an overall factor of about 2.3 compared to the CA baseline.

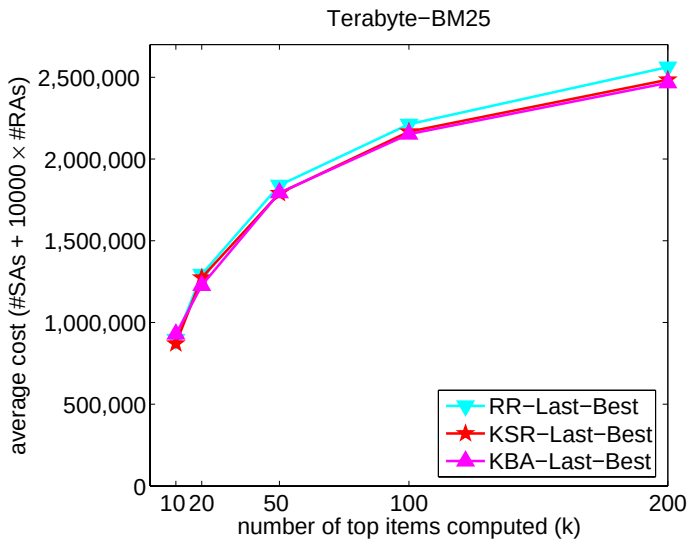


Figure 9.14: Average cost for the different SA scheduling approaches for Terabyte with a BM25 model, for varying k.

Varying the Query Size

In the next setup, we increase the query size m for the Terabyte setting by also taking terms from the TREC topic descriptions into account, i.e., we increase the average query size from $m = 2.9$ to $m = 8.3$ with a maximum of $m = 15$ terms which roughly simulates a query expansion task – a common technique in IR. Increasing the query dimensionality m yields further performance gains of up to a factor of 2.3 over NRA and a factor of 4 over CA. Note that NRA and CA essentially scan the whole lists for the larger m; then NRA has essentially the same costs as the Full Merge, while CA costs almost twice as much, due to its proportional number of random accesses.

Varying the c_R/c_S Ratio

By tuning the c_R/c_S ratio we can easily simulate different systems setups. Obviously, large ratios punish RAs and make the NRA or even the Full Merge more attractive. This is the case in systems with high sequential throughput and relatively low RA performance (e.g., $c_R/c_S = 10,000$ for mostly raw disk accesses with hardly any caching as opposed to $c_R/c_S = 100$ for a DBMS with lower sequential throughput but higher RA performance through caching). Figure 9.18 shows that for low values of c_R/c_S between 100 and 1,000, the combined scheduling strategies provide the highest cost savings with a factor of more than 2 for $k = 100$. Even when only very few RAs are allowed, a

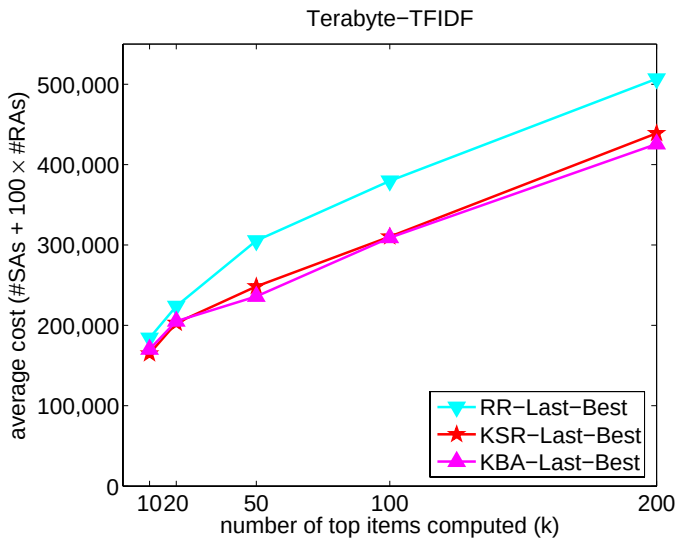


Figure 9.15: Average cost for the different SA scheduling approaches for Terabyte with a TF·IDF model, for varying k.

clever scheduling can still make a decisive difference and improve over NRA or Full Merge.

IMDB

The largest index lists derived from the IMDB collection (in the relational version) with up to a length of 285,000 entries are generated by the categorical attributes such as Genres and Years, whereas the largest inverted lists from text contents only yield a few thousand entries which are typically scanned through by the first block. This makes the collection provide an interesting mixture of short textual lists with quickly decreasing scores and longer lists of categorical values with a low skew and many score ties. Figure 9.19 shows that the performance gains here are a bit less than for Terabyte with a factor of 1.5 to 1.8 for $10 \leq k \leq 200$. For this particular combination of lists and mixture of score distributions, all top-k algorithms outperform the Full Merge baseline by a large margin, for wide ranges of k. Note that we are still able to stay very close to the lower bound compared to CA and NRA.

HTTP Worldcup Log

The HTTP Worldcup log yields highly skewed score distributions with a few users having downloaded up to 750 MB per day, whereas the average traffic per user and day lies between 50-100 KB. Figure 9.20 shows that CA

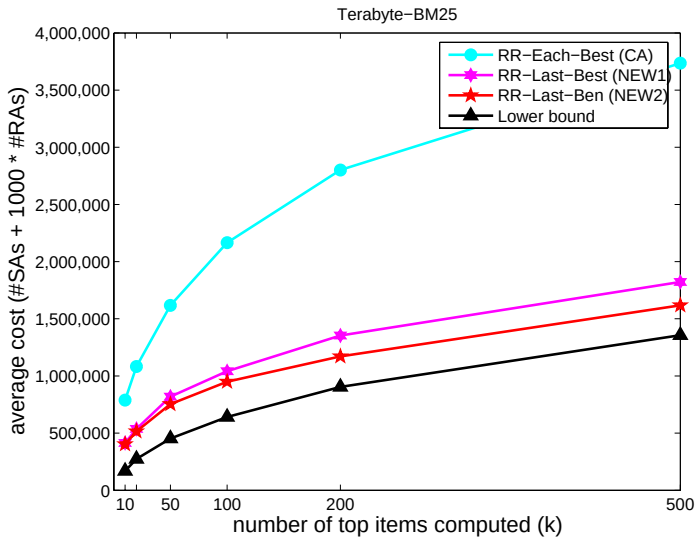


Figure 9.16: Average cost for the different RA scheduling approaches for Terabyte-BM25, for varying k.

(which is already close to optimal) becomes more competitive to our best algorithm (`KBA-Last-Ben` here) with only a factor of about 1.2 additional cost for k up to 100, because a few random accesses on the currently best-scored items typically suffice to yield the final top-k results. `KBA-Last-Ben` almost touches the lower bound for wide ranges of k. Note that for these skewed distributions, the benefit-optimized Knapsack `KBA` yields the better basis for SA scheduling. Also note that, here, NRA ends up scanning the full lists already for relatively small k.

Discussion of Index Access Scheduling Experiments

For many real-word data sets and score distributions, Fagin's originally proposed CA algorithm already yields a tough baseline. Except for extremely skewed distributions and small values of k, NRA is out of the question, because there is typically only a marginal difference between the final scores of the kth and $(k+1)$-ranked result which makes the best- and worstscores converge very slowly and leads to a very late threshold termination (Figure 9.12). On the other extreme, TA with its high overhead in random I/O is a viable choice only for setups with an extremely low c_R/c_S ratio. Our experiments demonstrate that our proposed algorithms perform much better than CA which is considered the most versatile variant of Fagin's algorithm, especially for larger k. According to Fagin's instance optimality proof [Fag02] which guarantees optimal access rates within a constant factor of $4m+k$ per

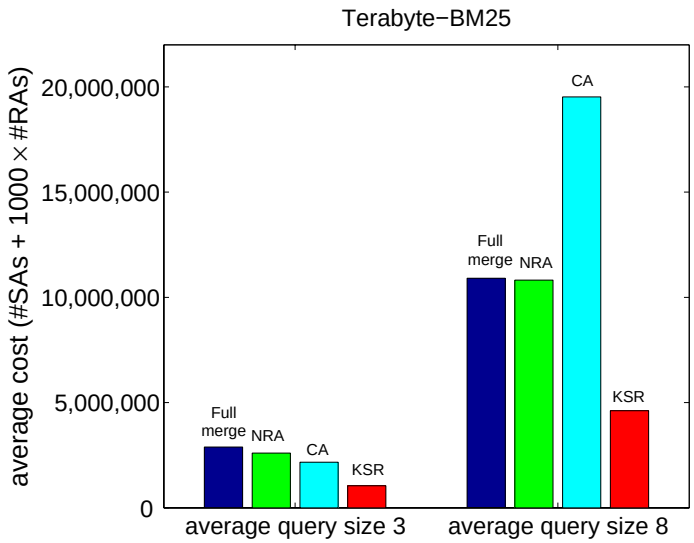


Figure 9.17: Average costs for Terabyte-BM25 of our best algorithm (`KSR-Last-Ben`) compared to various baselines, for shorter queries (left) and longer queries (right), for $k = 100$.

query, larger values of m and k are our best opportunity to beat this baseline by a very significant margin.

A comparison with two artificially generated *Uniform* and *Zipf* distributions for Terabyte reveals that for uniformly distributed scores, the round-robin SA scheduling already provides the best approach, whereas only for more skewed distributions (e.g., for TF·IDF, see Figure 9.15, or even Zipf) the Knapsack-based optimizations take effect. Fortunately, the Knapsack implementations tend to converge exactly to such a round-robin-like SA schedule in the Uniform case, hence they do not degenerate, but also can not improve much over the round-robin baseline in this case. Generally, a few judiciously scheduled RAs have the potential to yield an order of magnitude higher cost savings than the best SA scheduling could do.

For all setups, our algorithms that postpone random accesses to a late, more cost-beneficial phase and hence gather more information about the intermediate top-k and candidate items outperform their algorithmic pendants that eagerly trigger random accesses after each round or batch of sorted accesses (Figure 9.16). For all values of k and cost ratios c_R/c_S, our probabilistic extensions outperform the baseline algorithms by a large margin; moreover, they never degenerate or lead to higher access costs than their non-probabilistic counterparts. The simple Last-Probing approach with its heuristic stopping criterion is already a very solid basis; the cost-based Ben-

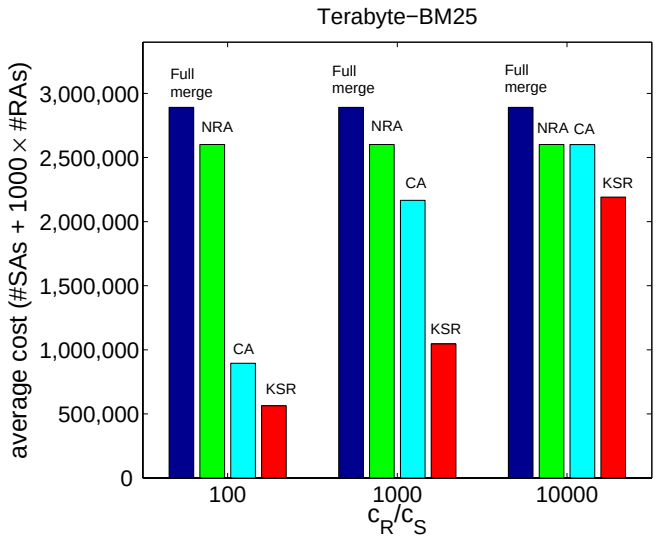


Figure 9.18: Average cost for Terabyte-BM25 of our best algorithm (`KSR-Last-Ben`) compared to various baselines with $c_R/c_S = 100$ (left), $c_R/c_S = 1,000$ (middle) and $c_R/c_S = 10,000$ (right), for $k = 100$.

Probing beats it merely by another 10 percent of costs saved and in fact comes close to the lower bound for many queries and collections (Figure 9.12, 9.19, and 9.20). Note that the iterative evaluation of the cost formulas in Sections 6.3, 6.4.1, and 6.4.2 is fairly light-weight so that the overhead of running the cost models for all candidates after a batch of b SAs is acceptable with regard to the costs saved in physical I/O.

9.7.3 Query Expansion

As for our query expansion runs, we again used Okapi BM25 for the per-term scores for both the Aquaint and Terabyte collections. Query expansion was based on WordNet concepts and the statistically quantified hypernym/hyponym relations in addition to the concept synsets as described in detail for the Independent Mapping WSD method in Section 7.2. We used both the benchmark query titles, which are merely 2 to 5 keywords, and the descriptions and narratives, which are a few sentences describing the query topic, to automatically extract term and phrase candidates for expansion.

Title terms were considered to be mandatory, with boosting weights $\beta_i = 1$ as described for the extended score aggregations in Section 3.5, terms from query descriptions were optional with $\beta_i = 0$; expansion terms were derived from both titles and descriptions and always considered optional. The similarity values α_i for the expansion terms (and phrases) were set according

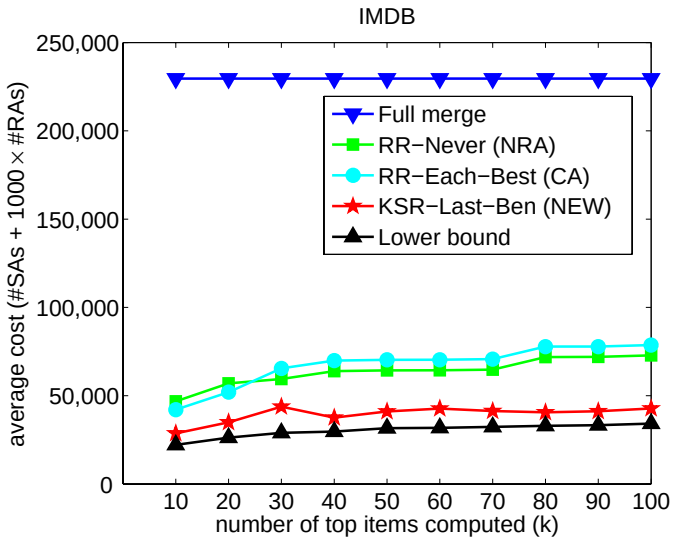


Figure 9.19: Average cost for IMDB of our best algorithm compared to various baselines and a computed lower bound, for varying k.

to the precomputed thesaurus-based Dice similarities. In the subsequent sections, we will mostly concentrate on results for the 50 hard queries of the Robust track data as these have become the gold standard for query expansion; we will discuss results for the Terabyte corpus thereafter with regard to scalability.

We compared both efficiency and effectiveness of the following methods:

- a *baseline* method without query expansion, with the option of probabilistic candidate pruning and variation of the control parameter ε (with $\varepsilon = 0.0$ being the conservative case where all speculative and approximative techniques are disabled),
- a *static expansion* method where we generate a large disjunctive keyword query by adding all expansion terms with similarity to at least one of the original query terms above a threshold θ (with $\theta = 1.0$ being the special case where only synonyms are added), and
- the *Incremental Merge* method for dynamic on-demand expansion, with an upper bound on the number of expansion terms controlled by θ.

With static expansion, the score aggregation function was the total sum of the scores from individual terms; with Incremental Merge, we used the max-score aggregation function introduced in Section 7.4.1. For both expansion methods we also varied the control parameter ε for probabilistic candidate

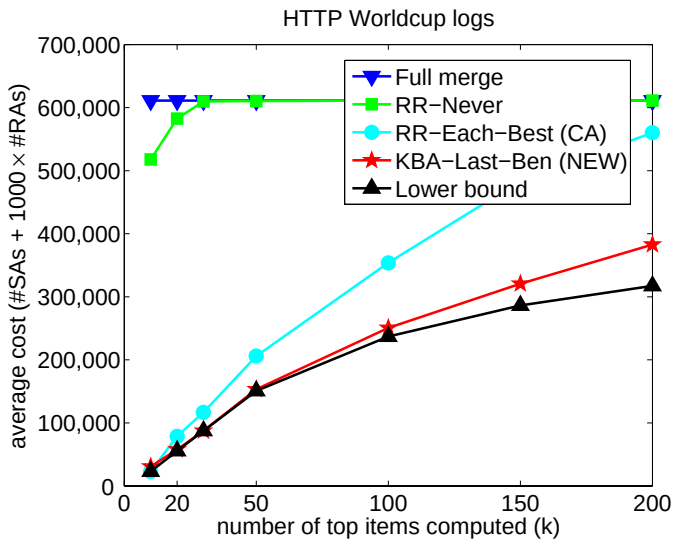


Figure 9.20: Average cost for the HTTP Worldcup logs of our best algorithm compared to various baselines and a computed lower bound, for varying k.

pruning. In addition to these methods under comparison, we also indicate some performance measures for a Full Merge query processing technique, where all inverted index lists that are relevant to a query are fully scanned and completely loaded into memory; this merely serves as a reference point. Note the recall-dependent MAP values were obtained from separate runs to obtain the top 1,000 for each query as demanded by TREC.

Further note that we decided to switch toward measuring runtimes in CPU seconds spent on the query processing, since CPU time is much less sustainable to caching effects, in particular for increasingly large expansions with huge and almost unpredictable variances in the wallclock runtimes of more than 300 percent (even with a "warm cache" when each query is directly executed twice in a role). Wallclock runtimes were generally about an order of magnitude higher than the reported CPU times.

Baseline Expansion Runs

Table 9.4 shows the results for the baseline run, i.e., without query expansion. The maximum query dimensionality was 4, the average query length was merely 2.5 terms; there was no consideration of phrases in this baseline runs.

We see that top-k query processing is generally much more efficient than the Full Merge technique with full index scans could possibly be, in terms of the number of sorted accesses to inverted index lists and the memory consumption. For the expansion setup, we did actually execute the Full

Merge, but merely measured the index-scan work (in the total amount of index list lengths) as this yields exactly the index access costs of this method. The rows for the top-k baseline method differ in their settings for the ε control parameter. Allowing a modest extent of probabilistic pruning led to noticeable performance gains, while staying at almost the same level of effectiveness.

We also see that enforcing conjunctive query evaluations in these benchmark settings is generally not beneficial for result quality. Already for the short title queries, the conjunctive run substantially loses in the recall-depending MAP metric with a drop from 0.091 to 0.068 (with $\varepsilon = 0$) compared to the default "andish" mode. Interestingly, also P@10 shows better results for andish, with a value of 0.252 for andish and a value of 0.244 for conjunctive queries. Note that expansion runs would not at all be feasible with a conjunctive setup.

In addition to P@10, MAP, and the relative precision, we also measured various other kinds of metrics relative to the query result quality of the conservative case with $\varepsilon = 0.0$, like the score error in the top-k lists or the footrule distance between ranks in the approximate and the "exact" top-k results. All these measures indicate that the probabilistic pruning affects the result quality only to a minor degree that would be acceptable by most applications. This makes the Prob-k method an excellent choice for a system where high precision at the top ranks is required and efficiency is crucial.

	ε	Max(m)	#SA	#RA	CPU	KB	P@10	MAP	prec
	Robust Baseline Runs								
Full Merge		4	2,305,637	0	n/a	13,509	0.25	0.09	1.00
Title-Only	0.0	4	1,439,815	0	9.4	519	0.25	0.09	1.00
andish	0.1	4	1,339,756	0	9.3	520	0.25	0.09	0.93
Title-Only	0.0	4	1,411,815	0	9.2	733	0.24	0.07	1.00
conjunctive	0.1	4	1,322,874	0	8.9	734	0.24	0.07	0.94

Table 9.4: Baseline runs for the 50 hard topics comparing conjunctive versus "andish" query evaluations (using only title keywords).

Different Expansion Strategies

The second part of Table 9.5 shows the efficiency and effectiveness results for the 50 hard queries of the TREC Robust track, comparing the Incremental Merge method with static query expansion.

Using the Independent Mapping with a rather conservative expansion strategy using only synonyms and first-order hyponyms of noun-phrases (i.e., directly related, more specific concepts) from the topic titles and descriptions

already produced fairly high-dimensional queries, with up to $m = 118$ terms (many of them marked as phrases); the average query size for these queries was $m = 35$ terms.

Compared to the baseline without query expansion, all expansion techniques significantly improved the result quality in terms of P@10 and MAP. The quality is still below the values reported by the very best TREC runs 2004 [Kwo04, LLYM04] which achieved about 0.37 in P@10, but the results are decent. Recall that our basis for query expansion, WordNet, is certainly not the most suitable choice for ad-hoc query expansion (at least not unless combined with other sources and techniques), and the emphasis of our work is on efficiency with good (but not eagerly hand-tuned) effectiveness.

The best result quality in our experiment, 0.31 in P@10, was achieved by the Incremental Merge technique with ε set to 0.0. Probabilistic pruning with $\varepsilon = 0.1$ reduced the number of sorted accesses by about 25 percent, but in terms of runtime (in CPU seconds) it gained a factor of 2 as it also incurred fewer random accesses (mostly for phrase matching) and lower memory overhead. For the static expansion technique, the cost savings by the probabilistic pruning were much higher, more than a factor of 4 in all major efficiency metrics, but this came at the expense of a significant loss in query result quality.

Notwithstanding this trade-off, this technique may be of interest for some applications. In terms of runtime, however, both Incremental Merge and static expansion performed almost equally well when probabilistic pruning was used. The absolute performance numbers of less than 2 seconds per query, with an academic prototype in Java and the high overhead of JDBC sessions, are very encouraging. It is particularly remarkable that for the Incremental Merge method, the probabilistic pruning affected the effectiveness only to a fairly moderate degree, reducing P@10 from 0.310 to 0.298. This seems to be a very low price for a speed-up factor of 2.

Figures 9.21 and 9.22 show how the effectiveness and efficiency measures change as the control parameter ε is varied from 0.0 and 0.1 toward higher, more aggressive values (with the extreme case 1.0 corresponding to a fixed amount of index-scan work, looking only at the $b = 500$ highest score entries of each index list). We see that the relative precision (prec) of most variants drops linearly with ε, which is just the expected behavior (see Section 5.4), but in terms of objective result quality, as measured by the TREC relevance assessments, the pruning technique performed much better: both precision@10 and MAP decrease only very moderately with increasing ε. For example, for $\varepsilon = 0.5$ the best Incremental Merge method could still achieve a precision@10 of about 0.27 while its runtime cost, in terms of sorted accesses, was reduced by a factor of more than 3.

	ε	Max(m)	#SA	#RA	CPU	KB	P@10	MAP	prec
	Robust Baseline Runs								
Full Merge		4	2,305,637	n/a					
Title-Only	0.0	4	1,439,815	0	9.4	431	0.25	0.09	1.00
	0.1	4	1,339,756	0	9.3	432	0.25	0.09	0.93
	Robust Fixed Expansions								
Full Merge		118	20,582,764	n/a					
Static Exp.	0.0	118	18,258,834	210,531	245.0	37,355	0.29	0.11	1.00
	0.1	118	3,622,686	49,783	79.6	5,895	0.24	0.09	0.54
Incr. Merge	0.0	118	7,908,666	53,050	159.1	17,393	0.31	0.12	1.00
	0.1	118	5,908,017	48,622	79.4	13,424	0.30	0.11	0.79
	Terabyte Fixed Expansions								
Full Merge		119	581,307,315	n/a					
Static Exp.	0.0	119	360,811,608	973,188	18,090.1	783,273	0.23	0.08	1.00
	0.1	119	63,028,142	120,180	6,785.5	101,333	0.22	0.07	0.63
Incr. Merge	0.0	119	87,327,339	211,981	10,734.3	575,440	0.27	0.10	1.00
	0.1	119	64,548,694	147,238	7,966.5	540,845	0.27	0.10	0.71

Table 9.5: Baseline and fixed expansions for the 50 hard Robust and the 50 Terabyte queries.

In Figures 9.21 and 9.22 the curves for static expansion with query titles and descriptions also reveal that the score predictor degenerates for very high-dimensional disjunctive queries because of neglecting feature correlations. This led to the sudden drop in the number of sorted accesses already for ε being as low as 0.1, but this came at the expense of a significant loss in retrieval quality. This phenomenon did not occur for the Incremental Merge method, where expansions are grouped into multiple nested top-k operators, such that the maximum number of query dimensions at the top level was only 22 compared to 118 for the static expansion (including phrases).

Figures 9.23 and 9.24 show the efficiency and effectiveness results as a function of varying the θ parameter, i.e., the aggressiveness of generating candidate terms and phrases for query expansion. Table 9.6 yields detailed figures for various combinations of θ and ε values. The charts demonstrate the robustness and self-tuning of the Incremental Merge method: even with very low θ, meaning very aggressive expansion, both P@10 and MAP values stayed at a very good level. The execution cost did significantly increase, however, but this is not surprising when you consider that the expansion candidate sets for some queries had up to $m = 876$ terms or phrases (at an average of $m = 230$ terms per query) – quite a stress test for any search engine. In combination with moderate probabilistic pruning, the Incremental Merge method was still able to answer all 50 queries in a few minutes, less than 4 seconds per query.

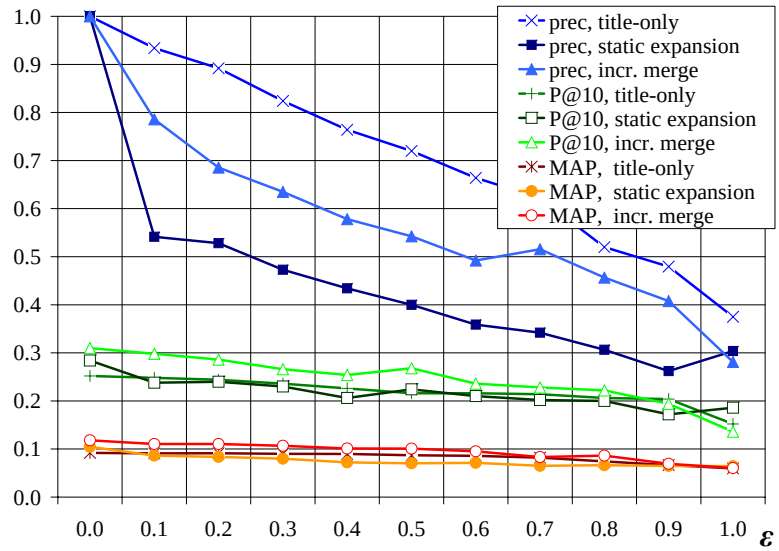


Figure 9.21: Expansion precision for the 50 hard Robust queries as a function of ε.

Scalability of Expansion Strategies

The evaluation on the Terabyte corpus and queries served as a proof of scalability for the developed top-k query processing and expansion methods. The third part of Table 9.5 shows the results of some of our runs, using the same fixed expansion technique as aforementioned for the Robust track. The performance gap for static expansions without versus the one with probabilistic pruning indicates the same overly aggressive behavior of the score predictor as for the high dimensional queries in the Robust setup – however, again at the expense of retrieval quality. Generally, expansion queries on Terabyte are in the order of up to several minutes per query rather than seconds. Moreover, we see that memory consumption becomes critical, with up to 783 MB for the static expansion method and 540 MB for Incremental Merge, respectively.

The overall efficiency gain in terms of access rates for the Incremental Merge method compared to the static expansion without probabilistic pruning is even more impressive by a factor of more than 4 and a factor of 8 compared to Full Merge, respectively, while achieving higher P@10 and MAP values. This makes the Incremental Merge approach the method of choice in terms of both retrieval robustness and efficiency.

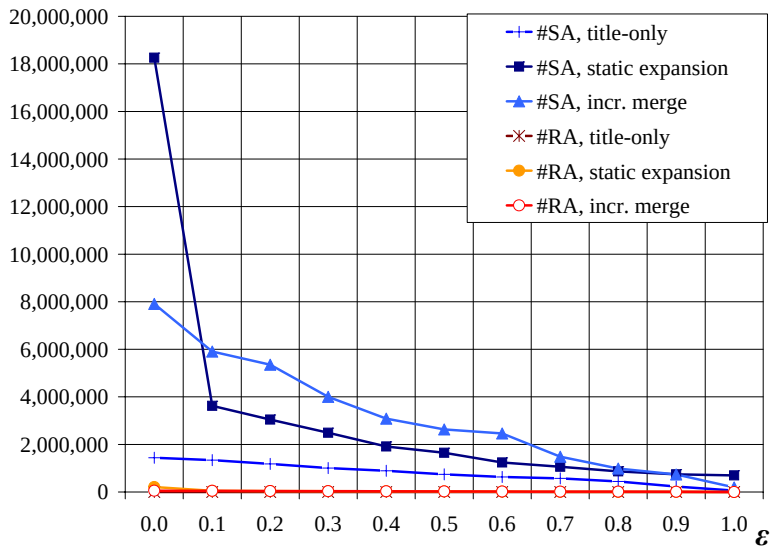


Figure 9.22: Expansion efficiency (in #SA and #RA) for the 50 hard Robust queries as a function of ε.

Discussion of Query Expansion Experiments

Incremental Merge clearly has the potential to outperform traditional static expansion in terms of both effectiveness and efficiency by exploiting the explicit structure that is given by an expanded query (i.e., the relation between original query concepts and expanded terms and phrase). It proved that it can achieve decent query-result quality while exhibiting very good execution cost and runtime behavior. Especially in combination with probabilistic score prediction and candidate pruning, it is a very efficient and effective, scalable method that could be of interest for industrial-strength search engines. Our emphasis has been on efficiency, so we used only a relatively simple basis for generating expansion terms and phrases. More elaborated expansion techniques, e.g., using document summaries from Google top-10 snippets or query associations from query logs along the lines of [Kwo04] or [LLYM04], could easily be supported by our algorithm, with similar expected trends for Incremental Merge as compared to static expansions.

9.7.4 TREC 2004

The following results are taken from our TREC 2004 and 2005 participation. The 2005 result were not previously published in an official TREC notebook

	θ	Max(m)	#SA	#RA	CPU	KB	P@10	MAP	prec
	Robust Large Expansions								
Full Merge	1.00	36	7,655,462	n/a					
	0.30	59	16,157,145						
	0.10	102	30,288,748						
	0.01	876	243,394,509						
Static Exp. $\varepsilon = 0.0$	1.00	36	5,427,347	169,635	38.5	8,987	0.28	0.11	1.00
	0.30	59	10,586,175	555,176	156.6	29,913	0.29	0.11	1.00
	0.10	102	15,541,754	1,950,718	230.8	60,195	0.27	0.11	1.00
	0.01	876	47,169,998	4,575,223	679.2	755,792	0.26	0.09	1.00
Incr. Merge $\varepsilon = 0.0$	1.00	36	4,850,129	60,739	33.6	6,802	0.28	0.12	1.00
	0.30	59	7,575,647	65,523	56.8	13,867	0.28	0.11	1.00
	0.10	102	10,889,717	77,261	100.1	25,628	0.27	0.10	1.00
	0.01	876	31,023,932	102,669	407.6	68,592	0.26	0.09	1.00
Incr. Merge $\varepsilon = 0.1$	1.00	36	3,668,119	30,759	26.2	3,257	0.28	0.12	0.65
	0.30	59	5,671,493	34,895	43.8	7,931	0.28	0.11	0.65
	0.10	102	7,615,389	38,641	78.3	24,454	0.28	0.10	0.67
	0.01	876	16,748,953	73,153	189.4	46,456	0.28	0.10	0.66

Table 9.6: Various θ expansions for the 50 hard Robust queries.

paper, because they provided the major intermediate step toward our final query expansion methods applied and published in [TSW05a]. For the three conceptually different tasks, the full range of our relevance scoring model as described in Chapter 3 came into play. Except for the 2005 Efficiency task, efficient query executions are only a minor aspect for the TREC runs, as generally the top 1,000 results per run were to be submitted to TREC to be able to ensure a sufficiently large recall base. Although TopX did not rank among the top participants in any of the tracks, we believe that our runs did a thorough job and that nethertheless the evaluation methodology applied throughout the whole thesis greatly benefits from these TREC participations. Note that TREC tasks are quite challenging and competitive, with many groups frequently participating over many years and hand-tuning systems for specific tasks.

Robust Track - Individual Retrieval Robustness

We submitted 10 runs for the 2004 Robust track with slightly different parameter settings and expansion strategies. The emphasis of this track was on the fine-tuning of our expansion methods and extracting additional query terms and phrases out of the extended description and narrative fields. We were using a TF·IDF model with boosting for the title keywords and additional, non-boosted keywords extracted from the description and narrative fields. Phrases were automatically detected by WordNet lookups, disambiguated using the Independent Mapping approach, and carefully expanded using only synonyms and first-order hyponyms. Query expansions using the new Incremental Merge method were conducted for noun-phrases only (as matched and disambiguated in WordNet), and only if they were sufficiently

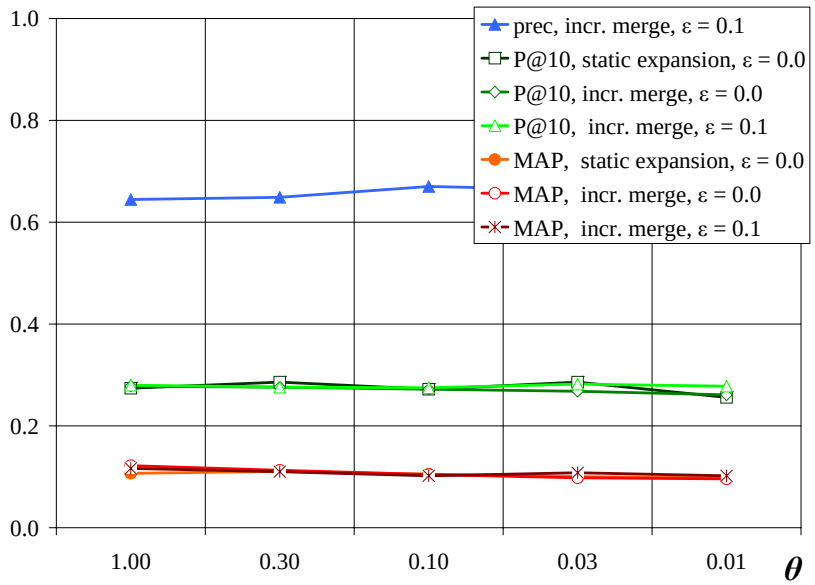


Figure 9.23: Expansion precision for the 50 hard Robust queries as a function of θ.

specific, i.e., with a high IDF value.

For the whole set of 249 Robust queries, we achieved good P@10 and MAP values of 0.37 and 0.17 (and even 0.42 for P@5), respectively, as shown in Figures 9.25 and 9.26; though we did not rank among the very best participants with up to 0.55 and 0.33 in P@10 and MAP, respectively. Figure 9.27 depicts an interesting plot that displays the relative difference of the TopX runs to the median average precision per query sorted by difference. We see that for many queries, we are in fact slightly above or at the median, but we also significantly drop below the median for about 50–60 percent of the queries. Note that the median displays a synthetic run, thus using a macro-average of all the achieved MAP values of all participants *per query*, i.e., none of the systems is necessarily always above or below that median for all queries.

Web Track - Topic Distillation & Named Page Search

For the Web track, we exploited the full range of our Web-specific extensions and the combined scoring approach described in Section 3.3 to get a smoothed influence of PageRank on the content scores. We submitted 5 runs with different parameter setups for studying the influence of the link struc-

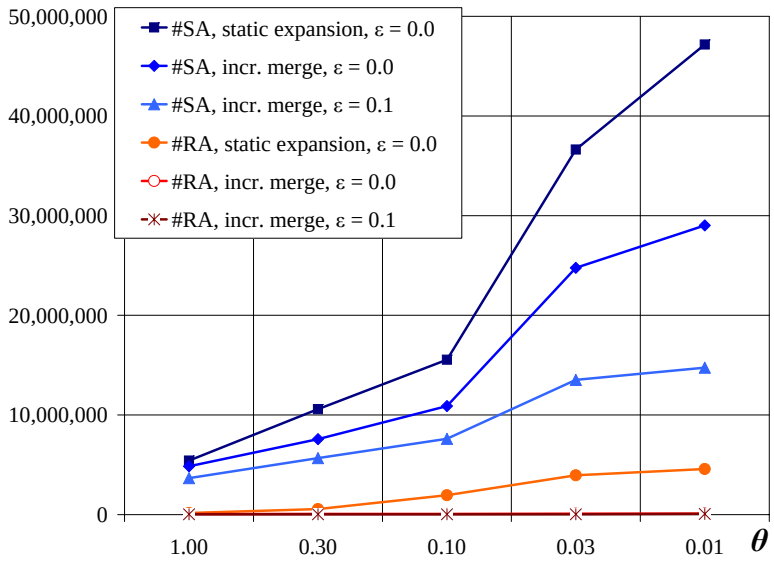


Figure 9.24: Expansion efficiency (in #SA and #RA) for the 50 hard Robust queries as a function of θ.

ture, anchor texts, and multiple weighted HTML fields. The 2004 Web track was a mixed query task with 75 home page finding queries, 75 named page finding queries, and 75 topic distillation queries (which were not explicitly distinguished by the topic statements). The goal was to find ranking approaches which work well over all the 225 queries, without having access to the actual query type labels.

Figure 9.28 shows a mean reciprocal rank (MRR) of 0.42 for the named page (NP) task which means that the named page was typically returned at ranks between 1 and 5 for most queries; and Figure 9.29 shows a very homogeneous flow of the precision/recall curve which makes this a very successful result from a user perspective. Figure 9.30, however, shows that we are still performing significantly below the (artificial) median run. Figure 9.43 shows a nice plot that compares the MRR results of all the participating groups.

Terabyte – Ad-hoc Retrieval on a Very Large Collection

For the Terabyte task, Figure 9.44 shows that we are competitive in terms of indexing and querying times compared to other participants. Indexing took about 14 hours for parsing the 426 GB Terabyte collection using three server machines in parallel (one of them being the database storage server) which confirms to a total of about 42 hours, and the average query response

Robust track results — Max-Planck-Institute for Computer Science

Summary Statistics	
Run ID	mpi04r07
Run Description:	Automatic run, title+desc+narr
Kendall τ of difficulty predictions:	-0.174

	Topic set			
	Old 200	New 49	Hard 50	All 249
Total number retrieved	49842	12243	12445	62085
Total relevant	15350	2062	4416	17412
Total relevant retrieved	4530	788	837	5318
Mean average precision	0.1723	0.1885	0.0629	0.1755
% topics with no relevant in top 10	16.0	18.4	26.0	16.5
area under MAP(X) curve	0.0034	0.0037	0.0009	0.0032

Document Level Averages				
	Precision			
	Old	New	Hard	All
At 5 docs	0.4200	0.3918	0.2280	0.4145
At 10 docs	0.3650	0.3469	0.2040	0.3614
At 15 docs	0.3330	0.3170	0.1973	0.3299
At 20 docs	0.3008	0.2929	0.1800	0.2992
At 30 docs	0.2582	0.2605	0.1673	0.2586
At 100 docs	0.1514	0.1357	0.1110	0.1483
At 200 docs	0.1036	0.0766	0.0776	0.0983
At 500 docs	0.0453	0.0322	0.0335	0.0427
At 1000 docs	0.0226	0.0161	0.0167	0.0214
R-Precision				
Exact	0.2304	0.2585	0.1194	0.2359

Figure 9.25: Result summaries for the best TopX Robust run 2004.

times were in between 6–9 seconds per query which provides acceptable run times even with no approximations being used. These runs also used our combined scoring approach described in Section 3.3, but with a strongly damped influence of the PageRank component compared to the Web track runs. Generally, we found that the extended Terabyte topic statements are much more similar to the Robust topics than to the Web track, since mostly content-oriented retrieval is required.

Figures 9.31 and 9.33 show similar trends of the best TopX run (out of 5 submitted TopX runs) as in the previous two tracks when compared with the result quality achieved by all participants, with a P@10 value of 0.36 and a MAP value of merely 0.03 (because we did not return the top 10,000 documents as actually demanded by the Terabyte track). Figure 9.33 shows that we are very close to the median for almost all the queries.

Note that with the runtime figures reported for our scheduling experi-

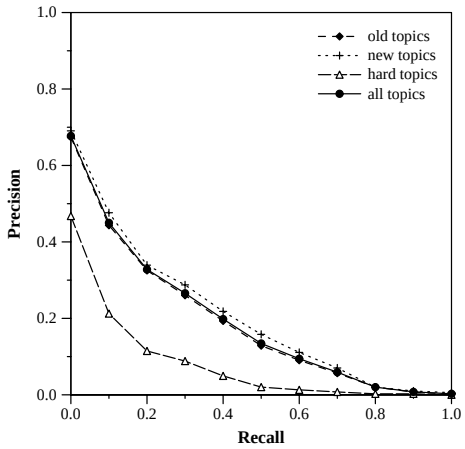


Figure 9.26: Precision/recall plot of the best TopX Robust run 2004.

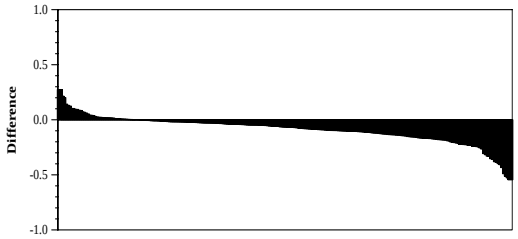


Figure 9.27: Difference to the median of the best TopX Robust run 2004.

ments (see Section 9.7.2) of our latest prototype implementation, we would rank at the very top among all Terabyte participants 2004 as depicted in Figure 9.44.

9.7.5 TREC 2005

Terabyte Efficiency Task

The TREC 2005 Terabyte Efficiency task used 50,000 queries from a commercial (but anonymous) search engine, so the challenge for our setup was to find a proper efficiency versus effectiveness trade-off. While the collection remained unchanged and we achieved similar indexing times for the Terabyte 2005 Efficiency task, we managed to press down querying time to an 0.3 seconds average per query (and 0.2 seconds for another run with slightly worse MAP and P@10 results) with the TopX Java prototype and Oracle as backend, using probabilistic pruning and our aggressive queuing strategy. The resulting P@10 and MAP values of 0.35 and 0.03 as shown in Figures 9.34 and 9.35 confirm our good experience for the probabilistic pruning and queue

Summary Statistics	
Run ID:	mpi04web08
Run Description	AnchorText-LinkStruct, DocStruct, URLLength-URLFeatures
Number of Topics:	225 (75 distillation, 75 named-page, 75 homepage)

Total number of documents	All topics	TD	NP	HP
Retrieved:	86353	28961	28746	28646
Relevant:	1763	1600	80	83
Rel-ret:	588	499	45	44

Official task measures	All topics	TD	NP	HP
Average Precision:	0.2860	0.0824	0.4101	0.3656
Success@1:	0.2978	0.2400	0.3467	0.3067
Success@5:	0.5200	0.5867	0.5067	0.4667
Success@10:	0.5644	0.6533	0.5467	0.4933
Precision @ 10 docs:		0.1427		
Precision @ "R" docs:	0.2540	0.1177	0.3511	0.2933
Mean repirocal rank:	0.3973	0.3899	0.4227	0.3793
Topics with no relevant found:	71	6	32	33

Figure 9.28: Result summaries for the best TopX Web run 2004.

management. Figure 9.36 shows that we dropped significantly below the median, however.

9.8 XML IR

For the XML experiments, we focus our attention on the INEX 2004 collection and queries, since they come shipped with official relevance judgments. To evaluate our runs, we chose the strict quantization, i.e., we are considering only those elements with a maximum specificity and exhaustiveness judgment of 3 for the recall base. Furthermore, we use a strict interpretation of both the support and target elements which further prunes the size of the recall base, i.e., the number of relevant elements referred to in the relevance judgments, thus anticipating the newly introduced SSCAS task in INEX 2005. Note that this strict notion of the query's support and target elements also naturally avoids overlap among result elements directly in the top-k engine, without requiring any additional query post-processing step.

9.8.1 Setup & Competitors

Our experiments for XML compared the following strategies for top-k query processing:

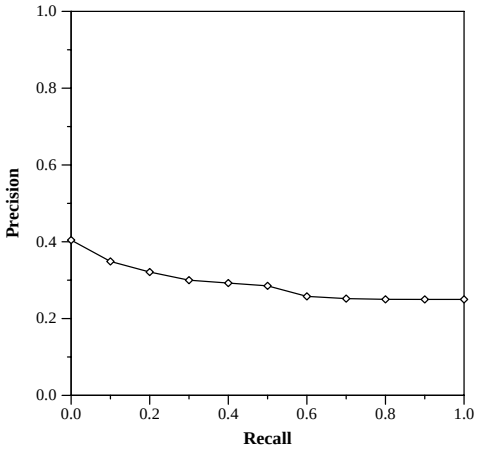


Figure 9.29: Precision/recall plot of the best TopX Web run 2004.

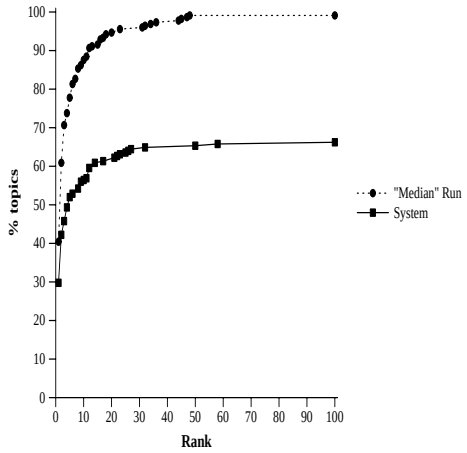


Figure 9.30: Difference to the median of the best TopX Web run 2004.

- TopX `Min-Probe` with the minimum probing strategy for random accesses as explained in Section 8.4.1.
- TopX `Ben-Probe` with the cost-based strategy for beneficial random accesses as explained in Section 8.4.2.
- TopX `Text`, using sorted-access-only to inverted term index lists for unstructured data. Here, random lookups are only used to clarify the final ranking among all top results' $[worstscore, bestscore]$ intervals.
- the `Full Merge` strategy where all query-relevant index lists are first completely fetched into main memory and the structural joins are performed in-memory with the full information about all candidates' struc-

Terabyte Track results — Max-Planck-Institute for Computer Scien

Summary Statistics	
Run ID	mpi04tb07
Run Type	automatic
Fields Used	title
Run Description	LinksUsed-NoAnchorTextUsed-DocStructUsed
Time to index	2520 min.
Time to retrieve top 20 per query	6 sec.
System RAM	2 GB
Size of on-disk structures	479 GB
Approx system cost	$15000
Number of Topics:	49
	Total number of documents over all topics
Retrieved:	48266
Relevant:	10617
Rel-ret:	4143

Recall Level Averages	
Recall	Precision
0.00	0.6559
0.10	0.3404
0.20	0.2499
0.30	0.1817
0.40	0.1102
0.50	0.0769
0.60	0.0583
0.70	0.0255
0.80	0.0104
0.90	0.0018
1.00	0.0000

Mean average precision	
non-interpolated	0.1249

Document Level Averages	
	Precision
At 5 docs	0.4245
At 10 docs	0.4122
At 15 docs	0.3864
At 20 docs	0.3663
At 30 docs	0.3388
At 100 docs	0.2667
At 200 docs	0.2033
At 500 docs	0.1306
At 1000 docs	0.0846

R-Precision: precision after R (number relevant) documents retrieved	
Exact	0.2108

Figure 9.31: Result summaries for the best TopX Terabyte run 2004.

ture using hash tables and cheap random access (which are then cheap because they do not incur any extra disk IO).

- `StructIndex`, the algorithm developed in [KKNR04] which uses a structure index to preselect candidates that satisfy the path conditions and then uses a TA-style evaluation strategy with random access to compute the top-k result.

- `StructIndex`$^+$, an optimized version of the `StructIndex` top-k algorithm, using also the extent chaining technique of [KKNR04] (see below).

In analogy to the text case, the Full Merge competitor corresponds to a traditional DBMS-style query processing that is extended by a structure-aware query processor for the XML case. It is inspired by the Holistic Twig

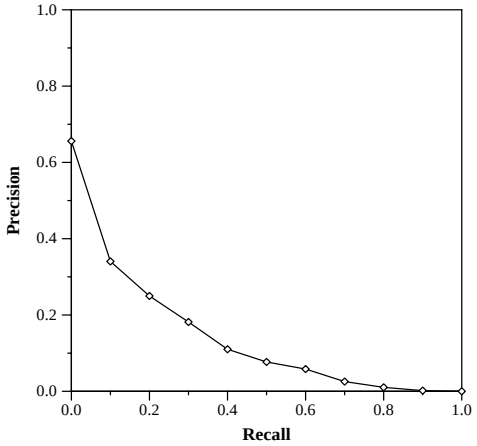


Figure 9.32: Precision/recall plot of the best TopX Terabyte run 2004.

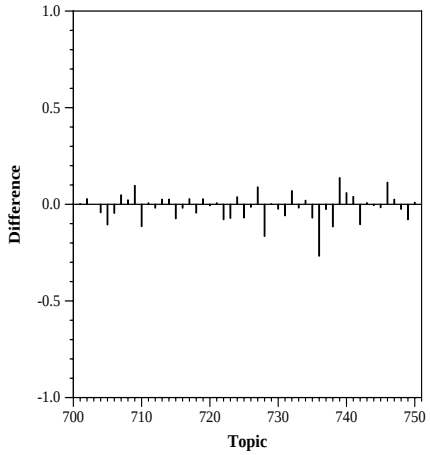


Figure 9.33: Difference to the median of the best TopX Terabyte run 2004.

Join of [BKS02, JWLY03, CMW03] which is probably the best known method for twig queries without any scoring or ranking (i.e., non-IR, Boolean XPath). It is a lower bound for the amount of index list accesses any non-TA-based implementation would have to make. For Full Merge, we report the MAP values taken from the top 1,000 final result elements (thus cutting off the long tail). The most interesting question here is of course, if the early termination of index scans that is characteristic for the top-k algorithms and for TopX really pays off in the end.

The `StructIndex` competitor is driven by the assumption that structural conditions are often quite selective. It uses a DataGuide-like structure index for first evaluating the structural skeleton of the query. This provides it with a compact representation of result candidates, namely, all element

Terabyte Track results, efficiency task — Max-Planck Institute

mpi05tbefcy1	
Run Type	automatic
Fields Used	title
Links/Anchor text/Structure used	doc struct
Number of Topics:	50
	Total number of documents over all topics
Retrieved:	1000
Relevant:	10407
Rel-ret:	312

% of collection indexed	100%	Number of CPUs	2
Indexing time	990 min.	System RAM	4 GB
Avg. time to retrieve top 20	0.310 sec.	Approx. system cost	$10000
Total processing time	15513 sec.	Year of system purchase	2004
Size of on-disk structures	600 GB		

Recall Level Averages	
Recall	Precision
0.00	0.5621
0.10	0.0805
0.20	0.0294
0.30	0.0044
0.40	0.0044
0.50	0.0044
0.60	0.0033
0.70	0.0033
0.80	0.0000
0.90	0.0000
1.00	0.0000

Mean average precision	
non-interpolated	0.0336

Document Level Averages	
	Precision
At 5 docs	0.3840
At 10 docs	0.3580
At 15 docs	0.3280
At 20 docs	0.3120
R-Precision	0.0439
bpref	0.0308

Figure 9.34: Result summaries for the best TopX Terabyte Efficiency run 2005.

combinations that satisfy all path constraints, concisely encoded into combinations of "extent identifiers". These identifiers are stored as additional attributes in the entries of the inverted index lists; so we can quickly test, if an element that is encountered in the index scan for a tag-term condition belongs to a document that satisfies the structure conditions. These tests are implemented by in-memory lookups to hash tables. As the algorithm also performs immediate lookups of missing tag-term scores (i.e., the original TA of Fagin [FLN03]), it generally follows an eager strategy for random accesses.

The optimized `StructIndex`$^+$ method assumes that all elements in a tag-term index list that have the same extent identifier in the structure index, i.e., have the same path tags from the root to the elements, form a forward-linked chain. The evaluation of the structure conditions then groups the resulting candidate elements by extent identifier, and conceptually invokes one sorted

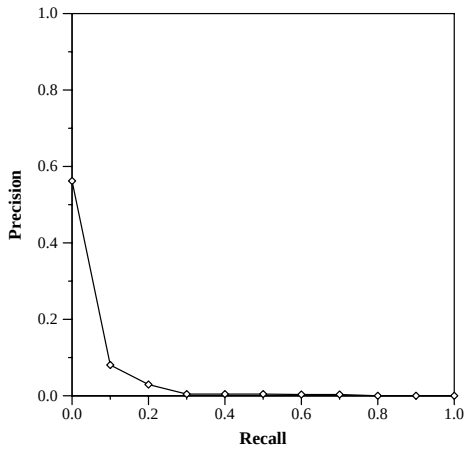


Figure 9.35: Precision/recall plot of the best TopX Terabyte Efficiency run 2005.

index scan for each extent. The implementation merges these cursors into one index scan that is mostly sequential but can perform skips. Depending on the gaps between index positions with candidates, this strategy may or may not degrade into a series of random accesses (if there are frequent and large skips), or it may or may not degrade into sequentially scanning all index entries anyway (if there are hardly any gaps). The two `StructIndex` approaches have been optimized to fit our *full-content* scoring model.

A comparison with Okapi BM25 scores for simple element contents has shown major benefits of our full-content scoring model in terms of result quality. Furthermore, `StructIndex` has been extended to also use a hash-based cache structure to avoid redundant random accesses. Note that typical INEX queries are already much more complex than the experiments in the original paper for `StructIndex` had considered.

9.8.2 Strict Content & Structure Queries

Although the TREC Terabyte collection is a pure text collection, we review runs for Terabyte in this section to compare query costs between two moderately large XML collections and a large text collection.

Table 9.7 presents detailed results comparing different TopX configurations and competitors for varying k for INEX, IMDB (in the semistructured version), and Terabyte. The table lists the following statistics each for the whole batch of queries: the parameter k, the amount of queries, the sum of sorted accesses #SA and the sum of random accesses #RA (both for the entire batch of queries), the the sum of CPU running times in seconds, the

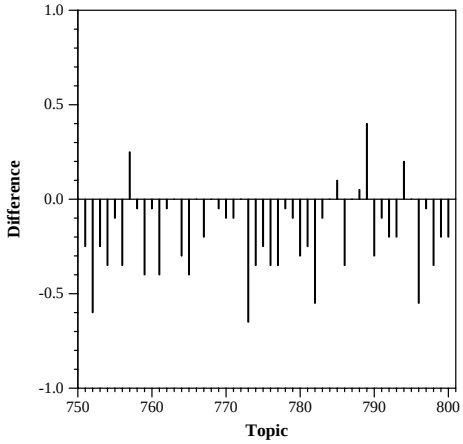


Figure 9.36: Difference to the median of the best TopX Terabyte Efficiency run 2005.

maximum memory consumption in KB, the macro-averaged absolute precision at k (P@k), and the mean average precision (MAP) for the top-k results using official relevance assessments for INEX and Terabyte. Since the IMDB is not an official benchmark setting, we omit the P@k and MAP values for the IMDB, although results were very good throughout the whole range of queries.

Wallclock times were typically a factor of 10 higher than CPU times (because of sequential and random disk I/O being performed by the Oracle server process), yielding user-perceived response times in the range of 0.1 to 0.7 seconds per INEX query (and up to 30 seconds for Terabyte) at $k = 10$ and without probabilistic pruning, and this proportion was fairly consistent for all algorithms and parameter settings.

Note that random accesses to test navigational query predicates are handled in the notion of expensive predicate probing to the external `TagsRA` element index which is not part of the actual inverted lists (see Section 8.4). That means that these random accesses are indispensable for query evaluations and cannot be compensated by a respective increase of sorted access rates. As such, we report the sorted and random access rates separately.

Baseline XML Runs

Table 9.7 shows that the conservative TopX method without probabilistic pruning ($\varepsilon = 0$) reduces the number of expensive random accesses by a factor of 50 to 80 compared to the `StructIndex` competitors on INEX in both the `Min-Probe` and `Ben-Probe` configurations, and still by a factor of

4 to 6 on IMDB, with very good rates of inexpensive sorted accesses (#Q denotes the number of queries in the different benchmark batches).

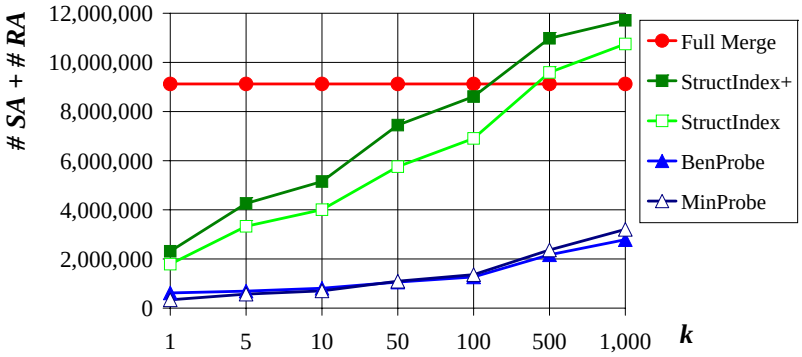


Figure 9.37: Efficiency (in $\#SA + \#RA$) as a function of k for INEX.

The `Min-Probe` scheduling outperforms `Ben-Probe` on INEX, in terms of saving random accesses. Random accesses have a very sensitive effect on running times, because they incur in an estimated runtime factor of about 20–50 in our specific hardware and database setup.

On IMDB, on the other hand, `Ben-Probe` was superior because of the different structural characteristics of the data. There are 106,970 distinct twig and 3,404 distinct path structures in INEX compared to only 3,859 distinct twigs and 111 distinct path structures in IMDB. Because of the high document-level selectivity and the deeper structure of path conditions on IMDB, the structural constraint tests using range scans on the `TagsRA` table to get the pre/postorder codings were much more expensive than for INEX so that random access scheduling became very crucial. Here, the cost-based `Ben-Probe` method showed its benefits. For the same reason, the `StructIndex` techniques became competitive to TopX for large k in terms of access rates, but for small k up to 100 both `Min-Probe` and `Ben-Probe` won by a large margin. Generally, sorted access shows its drawbacks for categorical attributes such as genres, birth places, etc., which yield very long lists with many ties in the local scores.

Note that the sorted-access cost of Full Merge is the same for all k values as the query-relevant index lists are completely fetched anyway (even for $k = 1$). We report the MAP value (which depends on k) for the top $k = 1,000$ in this case. For $k = 1,000$, the `StructIndex` approaches have even more index accesses than Full Merge because of the different index structures, where TopX still remains more efficient than Full Merge. Note that for k as large as 1,000 or higher, all top-k approaches increasingly degenerate.

The run Full-Merge-NR for INEX that is using a non-redundant scor-

	k	#Q	#SA	#RA	CPU	KB	P@k	MAP
	INEX							
Full Merge	1,000	46	9,122,318	0	12.4	14,981	0.03	0.17
Full-Merge-NR	1,000	46	8,189,566	0	11.3	14,981	0.01	0.07
StructIndex	1	46	289,160	1,500,804	7.4	3,351	0.57	0.04
	10	46	761,970	3,245,068	16.4	10,316	0.34	0.09
	100	46	1,966,960	4,938,645	29.8	12,810	0.13	0.15
	1,000	46	4,442,806	6,307,770	51.7	14,196	0.03	0.17
StructIndex$^+$	1	46	30,309	2,282,280	41.2	3,350	0.57	0.04
	10	46	77,482	5,074,384	84.5	10,316	0.34	0.09
	100	46	160,816	8,447,310	126.6	12,810	0.13	0.15
	1,000	46	271,803	11,441,431	168.3	14,197	0.03	0.17
TopX Ben-Probe	1	46	605,975	10,668	2.3	14,145	0.57	0.04
	10	46	723,169	84,424	3.6	14,317	0.34	0.09
	100	46	826,458	441,563	5.7	14,322	0.13	0.15
	1,000	46	882,929	1,902,427	16.2	14,370	0.03	0.17
TopX Min-Probe	1	46	322,109	15,876	0.7	11,263	0.57	0.04
	10	46	635,507	64,807	1.2	12,088	0.34	0.09
	100	46	999,608	361,706	2.9	13,335	0.13	0.15
	1,000	46	1,219,639	1,984,801	11.9	13,601	0.03	0.17
	IMDB (Semistructured)							
Full Merge	1,000	20	14,510,077	0	755.160	468,382	n/a	n/a
StructIndex	1	20	251,036	205,775	2.5	6,908	n/a	n/a
	10	20	346,697	291,655	3.3	8,417	n/a	n/a
	100	20	629,574	747,737	7.6	30,064	n/a	n/a
	1,000	20	1,274,624	1,735,399	20.1	39,559	n/a	n/a
StructIndex$^+$	1	20	15,250	208,140	0.2	6,908	n/a	n/a
	10	20	22,445	301,647	0.2	8,417	n/a	n/a
	100	20	67,065	782,856	0.4	30,064	n/a	n/a
	1,000	20	180,701	1,914,181	1.1	39,559	n/a	n/a
TopX Ben-Probe	1	20	202,429	8,672	1.3	40,726	n/a	n/a
	10	20	241,471	50,016	1.6	41,519	n/a	n/a
	100	20	248,080	187,684	2.5	39,549	n/a	n/a
	1,000	20	400,142	1,231,516	11.4	46,564	n/a	n/a
TopX Min-Probe	1	20	181,973	13,889	0.6	5,260	n/a	n/a
	10	20	317,380	72,196	2.4	8,683	n/a	n/a
	100	20	870,615	241,955	6.9	35,481	n/a	n/a
	1,000	20	993,751	1,326,999	24.3	82,942	n/a	n/a
	Terabyte							
Full Merge	1,000	50	105,806,358	0	n/a	n/a	0.08	0.13
TopX Text	1	50	13,452,578	1,390	91.5	6,259	0.16	0.01
	10	50	27,541,711	2,035	974.0	13,388	0.31	0.01
	100	50	53,000,119	3,192	4,879.2	16,735	0.21	0.07
	1,000	50	56,619,220	25,227	1,650.8	19,190	0.08	0.13

Table 9.7: Baseline runs for INEX, IMDB, and Terabyte, for various k.

ing model with Okapi BM25 weights on per-document statistics performs very poorly in terms of precision and MAP values for the top 1,000 results. This demonstrates the severe shortcomings of document-wide content scoring approaches in XML element retrieval. Note that MAP captures both precision and recall and is the key metric in the relevance assessment in both benchmarks, INEX and TREC.

Although there are much fewer *distinct* variations of path and twig structures for the IMDB than for INEX, the situation now changes to the benefit of the `Ben-Probe` scheduling. Despite the high structural selectivity of the IMDB collection, the conservative `Min-Probe` scheduler makes many random lookups to ancestor elements that would actually be dispensable, because,

e.g., name tags may not only appear in actor elements but in many different contexts like directors, producers, plot writers and various other person descriptions. This way, the `Min-Probe` approach yields slightly more overhead mostly in terms of random accesses. But also `Min-Probe` still outperforms the best `StructIndex`$^+$ competitor in terms of random access rates and a significant runtime gain especially for $k < 100$.

Again, Terabyte served as a stress test to our engine. The relatively high P@1,000 values indicate that the relevance sets are huge as well. Therefore the MAP values at the lower k mostly suffer from not returning the top 10,000 as originally proposed by TREC 2004 which we do not consider meaningful for a top-k engine (see also Sections 9.7.2 and 9.7.3 for more details on Terabyte).

9.8.3 Strict Content & Structure with Probabilistic Pruning

We also studied the influence of ε on performance and query result quality. The results for the `Min-Probe` scheduling are shown in Table 9.8. As an additional quality measure we again report the macro-averaged relative precision *prec* compared to the conservative algorithm with $\varepsilon = 0$.

	ε	#Q	#SA	#RA	CPU	KB	P@k	MAP	prec
	INEX								
TopX *Min − Probe*	0.10	46	426,986	59,414	2.9	13,071	0.32	0.08	0.80
	0.25	46	392,395	56,952	2.6	13,071	0.34	0.08	0.77
	0.50	46	231,109	48,963	1.2	9,582	0.31	0.08	0.65
	0.75	46	102,118	42,174	0.8	4,212	0.33	0.08	0.51
	1.00	46	36,936	35,327	0.3	758	0.30	0.07	0.38
	IMDB (Semistructured)								
TopX *Min − Probe*	0.10	20	250,173	57,066	0.6	8,683	n/a	n/a	0.95
	0.25	20	234,248	67,015	0.7	8,683	n/a	n/a	0.89
	0.50	20	147,471	55,197	0.5	8,683	n/a	n/a	0.80
	0.75	20	38,679	41,504	0.4	5,952	n/a	n/a	0.77
	1.00	20	10,068	37,058	0.3	942	n/a	n/a	0.78
	Terabyte								
TopX *Text*	0.10	50	23,010,990	1,106	215.6	13,391	0.27	0.01	0.73
	0.25	50	22,948,882	871	347.6	13,388	0.27	0.01	0.67
	0.50	50	19,283,550	993	160.9	13,388	0.26	0.01	0.60
	0.75	50	11,538,263	641	68.1	13,388	0.22	0.01	0.51
	1.00	50	151,043	779	0.6	41	0.19	0.01	0.35

Table 9.8: TopX runs with probabilistic pruning for various ε, for $k = 10$.

The probabilistic pruning reduces both the amount of index accesses and the overhead in queue operations, whereas the predictor overhead itself is almost negligible. The performance gain is another factor of 20 in access rates and a factor of 10 in runtimes compared to the conservative TopX and

more than two orders of magnitude compared to `StructIndex` or Full Merge throughout all the collections, still at very high precision values. Figure 9.38 shows performance gains for INEX, in terms of accesses rates, as a function of the ε value. Although there are minor reductions in the user-perceived quality measures like precision and MAP, probabilistic pruning hardly affects the result quality.

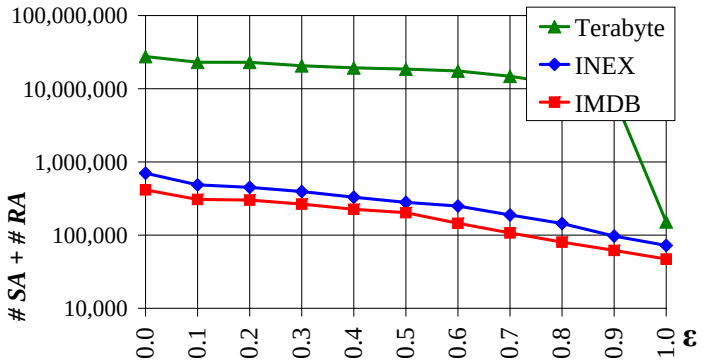


Figure 9.38: Efficiency (in $\#SA + \#RA$) as a function of ε for INEX, IMDB, and Terabyte, for $k = 10$.

Figure 9.39 shows that the relative precision value (prec) degrades at a much higher rate. This means that different results are returned at the top ranks, but they are equally good from a user perspective based on the official relevance assessments of INEX and TREC.

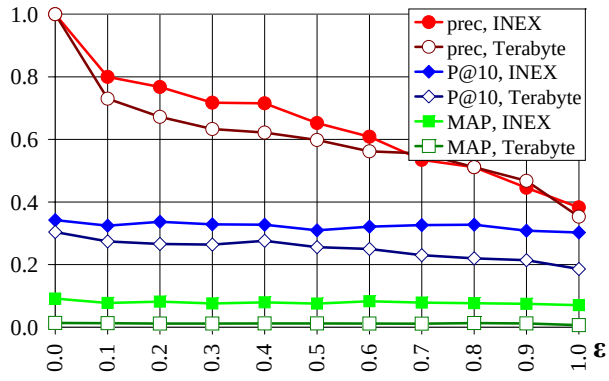


Figure 9.39: Precision as a function of ε for INEX and Terabyte, for $k = 10$.

9.8.4 INEX 2005

CO-Thorough

For the CO-Thorough task, Figure 9.40 shows that TopX ranks at position 22 for the nxCG@10 metric using a strict quantization with a value of 0.0379 and only at rank 37 of 55 submitted runs for MAP with a value of just 0.0008. As for all runs, we used the modified BM25 scoring model described above and also expensive text predicates to leverage phrases, negations, and mandatory terms.

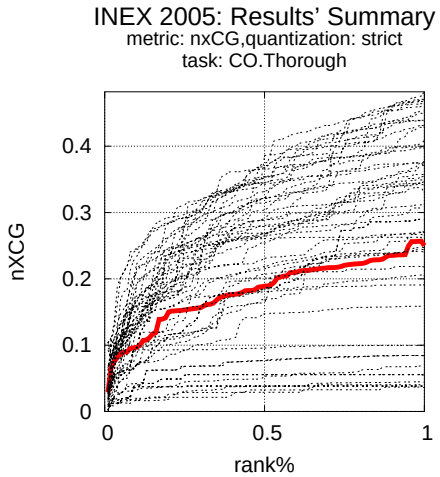


Figure 9.40: nxCG results for the TopX CO-Thorough run.

The very modest rank in the sensitive CO task attests that there is still some space for optimizations in our scoring model left for CO queries, when there is no explicit target element specified by the query (i.e., using the "//*" wildcard tag). Yet there was neither any restriction given on the result overlaps or granularities nor on the expected specificity or exhaustivity of special element types such as sections or paragraphs, such that the engine was allowed to return any type of element (also list items or even whole articles) according to their aggregated content scores. An additional simple post-processing step based on the element granularities and overlap removal would already be expected to achieve great performance gains here. However, for the old precision/recall metrics using the former INEX-eval tool with a strict quantization (as used in INEX '04), the TopX run ranks at a significantly better position of rank 3 with an average precision of 0.058 (MAP), which actually corresponds to the particular metric and setup for which we had tuned the system.

COS-Fetch&Browse

The situation improves for the COS-Fetch&Browse task, where the TopX run ranks at position 4 out of 19 with a value of 0.0601 in the ep-gr metric with strict quantization as shown in Figure 9.41. The high peak in first part of the ep/gr plot indicates that we were doing exceptionally well on the first ranks (i.e., the top-ranked results), but seem to have lost a substantial amount of recall at the lower ranks in this particular setting, however.

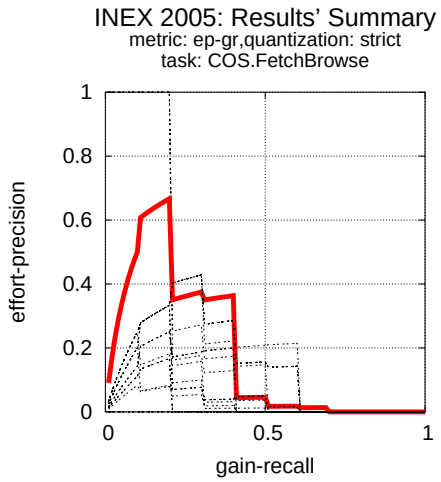


Figure 9.41: ep-gr results for the TopX COS-Fetch&Browse run.

TopX was configured to first rank the result documents according to their highest-ranked target element and then return all target elements within the same result document with the same score according to a strict interpretation of the target element given by the query which exactly matches our full-content scoring model. Here, the strict – XPath-like – interpretation of the query target element in combination with our full-content scoring model that treats each target element itself as a mini-document shows its benefits and naturally avoids overlap, since we return exactly the element type that is specified in the query and therefore seem to match the result granularity expected by a human user better.

SSCAS

Finally, the SSCAS task perfectly matches our strict interpretation of the target element with the precomputed full-content scores and no overlap allowed. The two submitted TopX runs rank at position 1 and 2 out of 25 submitted runs for the strict nxCG@10 metric with a very good value of 0.45

for both runs, and they still rank at position 1 and 6 for MAP with values of 0.0322 and 0.0272, respectively.

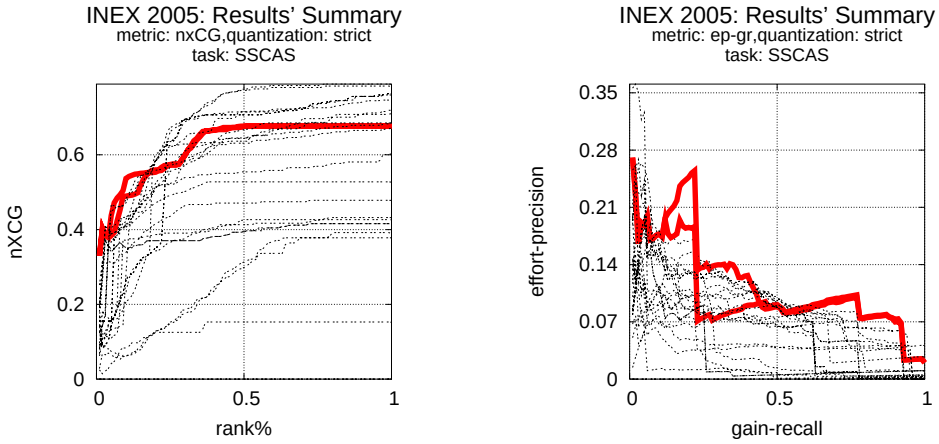


Figure 9.42: ep-gr and nxCG results for the two TopX SSCAS runs.

Figure 9.42 shows the nxCG and ep/gr plots for the two SSCAS runs, one with and one without considering expensive text predicates which performed almost equally well in this case. We see that TopX quickly reaches a maximum in the cumulated gain measure at about 50 percent of the returned ranks and then saturates, which is an excellent property for a top-k engine, because the best results are typically detected and returned at the first ranks already. Particularly nice is also the high peak of the second TopX run in the ep/gr metric which makes this run way outstanding from the competitors.

Although one might argue that this strict evaluation setup is less challenging from an IR point-of-view, the SSCAS task offers most opportunities to improve the efficiency of a structure-aware retrieval system, because the strict notion of all structural query components like target and support elements drastically reduces the amount of result candidates per document and, hence, across the corpus. Clever precomputation of the main query building blocks, namely tag-term pairs with their full-content scores, and index structures for efficient sorted and random access on whole element blocks grouped by document ids allows for decent runtimes of a true graph-based query engine that lies in the order of efficient text IR systems. Here, TopX can greatly accelerate query runtimes and achieve interactive response times at a remarkable result quality.

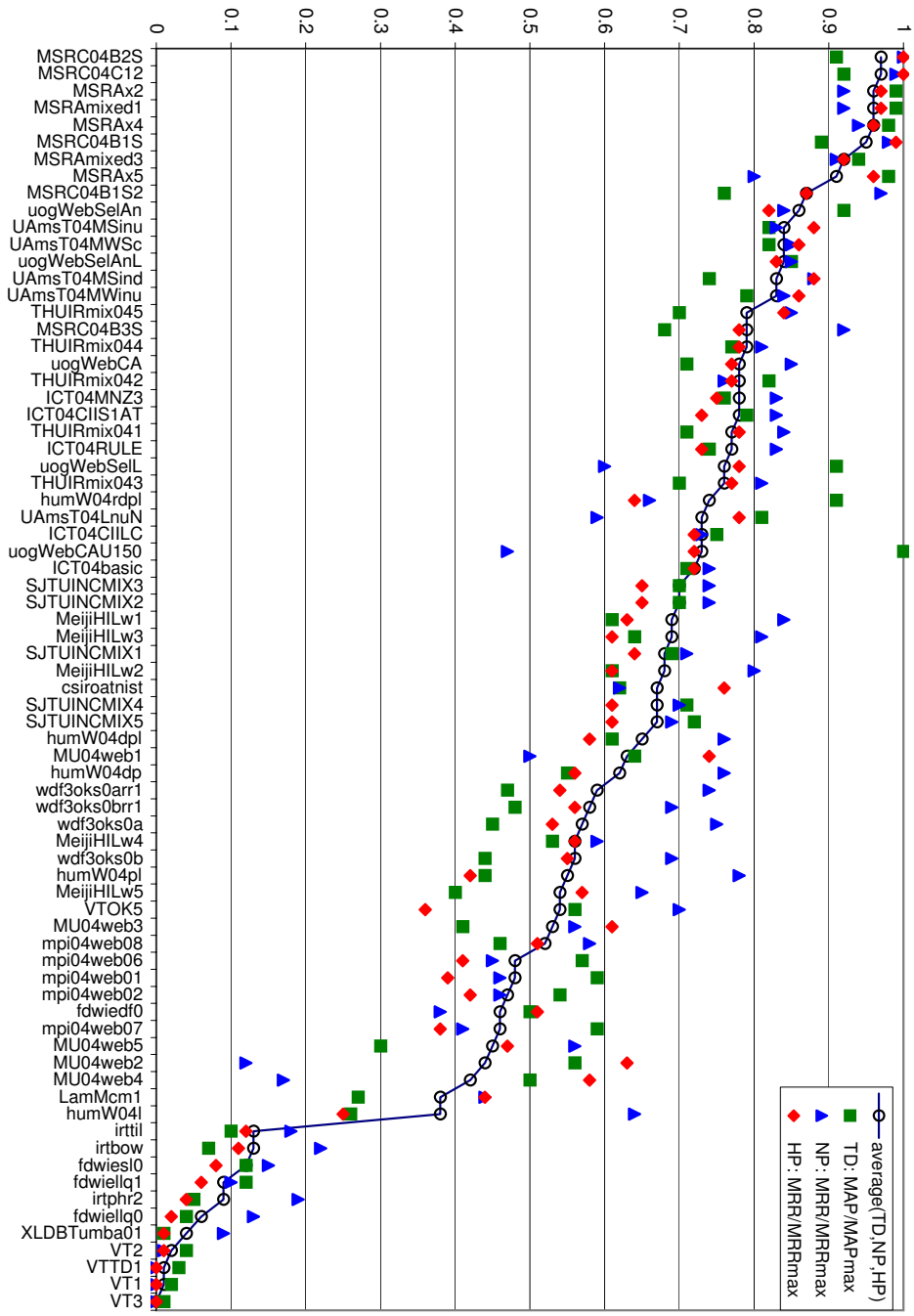


Figure 9.43: Mean Reciprocal Rank (MRR) results for all participants of the TREC 2004 Web track, broken down into average (AV), topic distillation (TD), named page (NP), and home page (HP) finding tasks. MPI/TopX runs are prefixed with 'mpi*'.

Group Id	Run Id	Topic Fields	Links?	Anchors?	Structure?	Query Time (s)	Index Time (h)	MAP
cmu.dir.callan	cmuapfs2500	TDN	N	N	N	600	20.0	0.284
	cmutufs2500	T	N	N	N	240	20.0	0.248
	cmutuns2500	T	N	N	N	75	20.0	0.207
dubblincity.u	DcuTB04Base	T	N	N	N	2	408.7	0.118
	DcuTB04Ucd1	TDN	N	Y	N	84	883.7	0.076
	DcuTB04Wbm25	T	N	N	Y	2	760.8	0.079
	DcuTB04Combo	T	N	Y	Y	2	906.0	0.033
	DcuTB04Ucd2	TDN	N	Y	N	15	457.5	0.070
etymon	nn04tint	T	N	N	N	25	44.8	0.112
	nn04eint	T	N	N	N	78	44.8	0.074
	nn04test	T	N	N	N	46	44.8	0.028
hummingbird	humT04l	T	N	N	Y	115	100.0	0.224
	humT04dvl	T	N	N	Y	142	100.0	0.212
	humT04vl	T	N	N	Y	119	100.0	0.221
	humT04l3	T	N	N	Y	49	100.0	0.155
	humT04	T	N	N	Y	50	100.0	0.196
iit	iit00t	T	N	N	N	23	8.0	0.210
	robertson	T	N	N	N	42	8.0	0.200
jhu.apl.mcnamee	apl04w4tdn	TDN	N	N	N	10000	0.0	0.034
	apl04w4t	T	N	N	N	10000	0.0	0.027
max-planck.theobald	mpi04tb07	T	Y	N	Y	6	42.0	0.125
	mpi04tb09	TD	Y	N	Y	9	42.0	0.123
	mpi04tb101	TD	Y	N	N	9	42.0	0.081
	mpi04tb81	TD	Y	N	N	9	42.0	0.092
	mpi04tb91	TD	Y	N	N	9	42.0	0.092
microsoft.asia	MSRAt3	T	N	Y	Y	1	11.6	0.171
	MSRAt4	T	N	Y	Y	1	11.6	0.188
	MSRAt5	T	N	Y	Y	1	11.6	0.190
	MSRAt2	T	N	N	Y	1	11.6	0.092
	MSRAt1	T	N	N	Y	1	11.6	0.191
rmit.scholer	zetbodoffff	T	N	N	N	25	13.5	0.219
	zetanch	T	N	Y	N	2	13.6	0.217
	zetplain	T	N	N	N	2	13.5	0.223
	zetfuzzy	T	N	Y	N	2	13.6	0.131
	zetfunkyz	T	N	Y	N	3	13.6	0.207

Group Id	Run Id	Topic Fields	Links?	Anchors?	Structure?	Query Time (s)	Index Time (h)	MAP
sabir.buckley	sabir04td3	D	N	N	N	18	14.0	0.117
	sabir04ta2	TDN	N	N	N	9	14.0	0.172
	sabir04tt	T	N	N	N	1	14.0	0.116
	sabir04td2	D	N	N	N	3	14.0	0.121
	sabir04tt2	T	N	N	N	1	14.0	0.118
tsinghua.ma	THUIRtb5	T	N	N	N	15	32.0	0.244
	THUIRtb4	TDN	N	Y	N	55	17.0	0.245
	THUIRtb3	T	N	Y	N	9	17.0	0.220
	THUIRtb2	TDN	N	Y	Y	18	2.8	0.056
	THUIRtb6	T	N	N	N	16	32.0	0.204
u.alaska	irttbtl	T	N	N	Y	5	30.0	0.009
u.amsterdam.lit	UAmsT04TBm1	T	N	Y	Y	90	4.3	0.044
	UAmsT04TBanc	T	N	Y	N	1	0.3	0.013
	UAmsT04TBm1p	T	N	Y	Y	90	4.3	0.043
	UAmsT04TBtit	T	N	N	Y	20	4.0	0.039
	UAmsT04TBm3	T	N	Y	Y	90	4.3	0.043
u.glasgow	uogTBQEL	TDN	N	N	N	46	200.6	0.307
	uogTBPoolQEL	TDN	N	N	N	46	200.6	0.231
	uogTBBaseS	T	N	N	N	4	200.6	0.271
	uogTBAnchS	T	N	Y	N	3	501.7	0.269
	uogTBBaseL	TDN	N	N	N	28	200.6	0.305
u.mass	indri04AWRM	T	N	N	N	39	5.9	0.284
	indri04AW	T	N	N	N	7	5.9	0.269
	indri04QLRM	T	N	N	N	26	5.9	0.253
	indri04QL	T	N	N	N	1	5.9	0.251
	indri04FAW	T	N	Y	Y	52	21.6	0.279
u.melbourne	MU04tb3	T	Y	Y	N	0.08	2.5	0.043
	MU04tb2	T	N	Y	N	0.08	2.5	0.063
	MU04tb4	T	Y	Y	N	0.36	13.0	0.268
	MU04tb1	T	N	N	N	0.08	1.7	0.266
	MU04tb5	T	Y	Y	N	0.08	2.5	0.064
upisa.attardi	pisa4	T	Y	Y	Y	3	16.0	0.103
	pisa3	T	Y	Y	Y	3	16.0	0.107
	pisa2	T	Y	Y	Y	3	16.0	0.096
	pisa1	T	Y	Y	Y	1	16.0	0.050

Figure 9.44: Indexing statistics of all participants of the TREC 2004 Terabyte track. MPI/TopX runs are prefixed with 'mpi*'.

Chapter 10

Conclusions

The TopX engine combines an extended TA-style algorithm with a focus on inexpensive sorted accesses with a number of novel features: carefully designed, precomputed index tables that help avoiding or postponing random accesses; a highly tuned method for index scans and priority queue management; probabilistic score predictors for early candidate pruning; dynamic query expansion with incremental merging of index lists; and nested top-k operators for phrase matching. The thesis presents a suite of novel techniques for efficient and versatile top-k query processing techniques and tested them on various real-world and official benchmark collections. We show that, already in the conservative mode with no approximations allowed, the engine outperforms previous approaches by a significant margin.

All components are seamlessly integrated in a versatile query processor and can be gradually plugged into the core engine. For example, the probabilistic candidate pruning and cost-based scheduling decisions use very similar assumptions on the underlying score distributions and can easily be combined as two complementary approaches that nicely supplement each other ("throw away the bad ones and look up the good ones"). Taking our very promising results for index access scheduling into account, which are consistently good over all tested data collections and queries, we consider the problem of scheduling for top-k query optimization as solved. As for the query expansion approach, the Incremental Merge technique clearly outperformed traditional methods with static expansions, and proved that it can achieve decent query-result quality while exhibiting very good execution cost and runtime behavior. Especially in combination with probabilistic score prediction and candidate pruning, it is a very efficient and effective, scalable method that could be of interest for industrial-strength search engines.

Our future work will generally aim to combine the best methods from structured XML querying and non-schematic IR. We believe that the research

area for top-k query processing is by far not exhausted but still opens a large variety of open issues and interesting challenges for future work; some of the most distinctive challenges are briefly picked in the following subsection.

10.1 Open Issues & Future Work

Index Compression & Scalability

The way our full-content index for XML data is organized basically introduces two dimensions of redundancy. The first factor for redundancy is inevitably obtained from the definition of our relational schema, i.e., through storing the pre- and postorder labels along with several redundant attributes per element and term entry. The second factor is related to the materialization of our scoring model and is obtained through the redundant full-content text nodes that we associate with each element. While we consider this to be a crucial part and basic building block of our scoring model that contributes to the (probabilistic) justification of our result ranking approach, the first issue could easily be optimized by any kind of index compression technique proposed in the literature. This would require moving away from the relational backend, however.

Graph-aware Top-k

The type of XPath evaluations we pursue in the current TopX implementation is tuned for IR-oriented queries and collections, with reasonable document sizes and queries consisting of mostly content-related conditions and a few location steps. The document boundaries principally bound the range of our in-memory structural joins, too, which are then very efficient, because they are mostly hash-based. While our algorithm already successfully deals with uncertainty in the structure of partly evaluated candidates to enable mostly sorted access to large, disk-resident index structure, a true graph top-k algorithm (e.g., querying a single 2-Gigabyte DBLP document) is still an open issue. In particular, the structural joins might suffer from significantly larger in-memory and on-disk range scans for large document partitions (e.g., using the pre/postorder labeling scheme). This might make the usage of R-tree-like index structures over these attributes attractive. Moreover, the efficient support for structured queries across large, interlinked document graphs seems very challenging.

Incorporating Non-Monotonous Score Aggregations

Non-monotonous score aggregations that would also take the proximity of terms in a keyword query or the compactness of an XML element "as a whole" into account are a particularly interesting issue from an IR-point-of view. Incorporating these functions in a true top-k-query processor would require the engine to keep the upper bestscore bounds of candidates more generous with regard to some additional scoring component the would be derived from this holistic score. It is not clear to what extend this would affect the performance of the engine; in the worst case this might prevent the worst- and bestscore bounds from converging and, thus, make the engine degrade to a full merge algorithm. Clever random access scheduling would probably be key for these extensions. Furthermore, one might try to precompute large, common query patterns such as frequent term pairs with significantly lower combined selectivity and thus even exploit positive effects for pruning when scanning these combined lists.

Instance Optimality

The class of algorithms that we investigate corresponds to the algorithmic paradigm of threshold algorithms (TA) that has been originally proposed and most intensively studied by Fagin in [FLN03]. In particular the Combined Algorithm (CA), with its basic cost model for random access scheduling, has been proven to be instance optimal for any given data and query within a constant factor of $4m + k$ in the query dimensionality m and the amount of top retrieved data items k.

Our empirical results of the refined index access scheduling strategies proposed in the present work – moreover the Last-Probing approach (see Section 6.4.1) that postpones all random access to a late, more cost-beneficial phase – strongly indicates we can keep the algorithm very close to the absolute lower cost bound for a wide range of k and in fact even up to a point where any top-k algorithm would degrade to a full merge. This might lead to a refinement of Fagin's instance optimality proof that might even abandon the k factor from the optimality bound which would of course require a more formal proof.

Implementation Issues

Our very latest efforts as indicated in the experiments section for the scheduling experiments, investigate the optimization of our implementation, including data structures, index organization, and merge-joining large list blocks

in-memory. While we already have a brand new C++ prototype that directly accesses inverted files with a highly specialized and compressed index organization, reimplementing the XML engine would be very time consuming. A potential cost factor of up to 20 as indicated for the text experiments, in particular for very large collections where many index lists are spanning multiple disk tracks, seems very attractive, however.

10.2 Concluding Remarks

TopX has been developed for two years now, and the initial, small proof-of-concept-like prototype has become a rather complex framework for indexing and querying all kinds of data collections. We believe that the resulting publications could achieve a noticeable contribution to this (still emerging) research area and try to further pursue some of the particularly interesting challenges as mentioned above.

As for the immediate future, TopX will be the official host for the INEX 2006 topic development phase for both the IEEE and the new Wikipedia XML collection, the latter extracted and annotated at *Laboratoire d'informatique de Paris 6* (LIP6), with more than 600,000 documents and more than 120,000 XML elements which makes the new collection a factor of about 10 larger than the previous IEEE collection. TopX is permanently online and the publicly available default engine (in the non-approximative mode) for the INEX topic development and the Interactive track.

Appendix A

APPENDIX

A.1 Database Tables & Index Structures (DDL)

A.1.1 Text Schema

Figure A.1 depicts the table and index definitions of the TopX text schema using Oracle 10g.

A.1.2 XML Schema

Figure A.2 depicts the table and index definitions of the TopX XML schema using Oracle 10g. Figure A.2 depicts the schema extension for the hybrid index structures approach described in Section 8.5.1.

```
// Document metadata
CREATE TABLE Documents (
  docid NUMBER(8), // Document identifier
  URI VARCHAR2(256), // Unified Resource Identifier
  ...  // Extensible list of document metadata
  CONSTRAINT Documents_PK PRIMARY KEY(docid)
) ORGANIZATION INDEX COMPRESS;
// Text index
CREATE TABLE TextFeaturesRA (
  docid NUMBER(8), // Document idenifier
  term VARCHAR2(32), // Term feature
  score NUMBER(7,6), // Local score
  CONSTRAINT TextFeaturesRA_PK PRIMARY KEY(docid, term)
) ORGANIZATION INDEX COMPRESS;
// Sorted access by term in descending order of score
CREATE INDEX TextFeaturesSA ON
  TextFeaturesRA(term, score DESC, docid) COMPRESS;
// Term offset index for phrase matching
CREATE TABLE TermsRA (
  docid NUMBER(8), // Document id
  term VARCHAR2(32), // Term feature
  position NUMER(6), // Term position in document
  CONSTRAINT TermsRA_PK PRIMARY KEY(docid, term, position)
) ORGANIZATION INDEX COMPRESS;
```

Figure A.1: TopX schema definitions for text index structures.

A.2 OpenMaple Scripts for Chernoff-Hoeffding Bounds

A.2.1 Chernoff-Hoeffding Bounds

Figure A.4 depicts an OpenMaple procedure to compute Chernoff-Hoeffding bounds for Uniform score distributions assuming feature independence.

A.2.2 Generalized Chernoff-Hoeffding Bounds

Figure A.5 depicts an OpenMaple procedure to compute generalized Chernoff-Hoeffding bounds for Uniform score distributions assuming limited feature independence.

```
// Full content index for tag-term pairs
CREATE TABLE TagTermFeaturesRA (
  docid NUMBER(8), // Document identifier
  tag VARCHAR2(32), // Element name
  term VARCHAR2(32), // Term in element
  pre NUMBER(6), // Pre-order of enclosing element
  post NUMBER(6), // Post-order of enclosing element
  score NUMBER(7,6), // Full-content score of tag-term pair
  maxscore NUMBER(7,6), // Max(score) of tag-term pair per doc
  CONSTRAINT TagTermsFeaturesRA_PK PRIMARY KEY(docid, tag, term)
) ORGANIZATION INDEX COMPRESS;
// Sorted access by (tag,term) in descending order of maxscore
CREATE INDEX TagTermFeaturesSA ON
  TagTermFeaturesRA(tag, term, maxscore DESC, docid, score DESC, pre, post) COMPRESS;
// Navigational element index
CREATE TABLE TagsRA (
  docid NUMBER(8), // Document identifier
  tag VARCHAR2(32), // Element name
  pre NUMBER(6), // Pre-order of element
  post NUMBER(6), // Post-order of element
  CONSTRAINT TagsRA_PK PRIMARY KEY(docid, tag, pre)
) ORGANIZATION INDEX COMPRESS;
// Term offset index for phrase matching
CREATE TABLE TermsRA (
  docid NUMBER(8), // Document identifier
  term VARCHAR2(32), // Term token
  position NUMER(6), // Term position in element
  pre NUMBER(6), // Pre-order of enclosing element
  post NUMBER(6), // Post-order of enclosing element
  CONSTRAINT TermsRA_PK PRIMARY KEY(docid, term, position)
) ORGANIZATION INDEX COMPRESS;
```

Figure A.2: TopX schema definitions for XML index structures.

A.3 Index Access Scheduling

A.3.1 NP-hardness of the Sorted-Access Scheduling Problem

The KNAPSACK decision problem in general can be briefly formulated as follows: Given m items X_i ($i = 1..m$), each with weight w_i and utility u_i, and a weight capacity C, decide for a given constant U if there is a subset $S \subseteq [1..m]$ such that the total utility is at least U, $\sum_{j \in S} u_j \geq U$, and the capacity constraint $\sum_{j \in S} w_j \leq C$ is satisfied. As usual, the solution to the optimization problem (i.e., maximize the total utility) can be derived from

```
 The following tables are required for the hybrid index approach only
// DataGuide index containing all distinct root-to-leaf paths in the collection
CREATE TABLE DataGuide (
  path VARCHAR2(1024), // Labeled path
  bucketid NUMBER(6),  // Compact bucket identifier for the labeled path
  CONSTRAINT DataGuide_PK PRIMARY KEY(path)
) ORGANIZATION INDEX COMPRESS;
// Navigational element index
CREATE TABLE BucketidsRA (
  docid NUMBER(8), // Document identifier
  bucketid NUMBER(6), // DataGuide-like bucket identifier
  pre NUMBER(6), // Pre-order of element
  post NUMBER(6), // Post-order of element
  CONSTRAINT TagsRA_PK PRIMARY KEY(docid, bucketid, pre)
) ORGANIZATION INDEX COMPRESS;
// Full content index for bucketid-term pairs
CREATE TABLE BucketidTermFeaturesRA (
  docid NUMBER(8), // Document identifier
  bucketid NUMBER(6), // DataGuide-like bucket identifier
  term VARCHAR2(32), // Term in element
  pre NUMBER(6), // Pre-order of enclosing element
  post NUMBER(6), // Post-order of enclosing element
  score NUMBER(7,6), // Full-content score of tag-term pair
  maxscore NUMBER(7,6), // Max(score) of tag-term pair per doc
  CONSTRAINT BucketidTermsFeaturesRA_PK PRIMARY KEY(docid, bucketid, term)
) ORGANIZATION INDEX COMPRESS;
// Sorted access by (bucketid,term) in descending order of maxscore
CREATE INDEX BucketidTermFeaturesSA ON
  BucketidTermFeaturesRA(bucketid, term, maxscore DESC, docid, score DESC, pre, post)
COMPRESS;
```

Figure A.3: TopX schema extension for hybrid index structures using DataGuides and Pre/Postorder labels.

the solution to the decision problem by a binary search over U.

Given an instance of KNAPSACK, we construct the following instance of the sorted-access scheduling decision problem SAS as follows. We consider m lists where the ith list has at its first $w_i - 1$ positions a constant score of 1 and thus score decrease 0, at position w_i a score decrease u_i (i.e., a resulting score $1 - u_i$, and subsequently the same constant score, i.e., no further score decrease. We claim that (A) a packing for this instance of KNAPSACK has capacity $\leq C$ and utility $\geq U$ if and only if (B) the corresponding SAS instance has a scan of total depth C and score decrease of $\geq U$.
Proof of $(A) \Rightarrow (B)$:
Given (A), we have $i_1, \ldots, i_k$ such that $w_{i_1}+...+w_{i_k} \leq C$ and $u_{i_1}+...+u_{i_k} \geq U$.

```
uniformbound := proc(delta, higharray, n);
  L := 1;
  for i from 1 to n do
    if higharray[i] > 0 then
      f1(x) := 1 / higharray[i];
      L1(s) := int(exp(-s * x) * f1(x), x=0..higharray[i]);
      L := L * L1(s);
    end if;
  end do;

  bound := exp(-s * delta) * subs(s=-s, L);
  limitbound := limit(bound, s=0, right);
  diffbound := diff(bound,s);
  eq := diffbound=0;
  mins := fsolve(eq, s);

  if mins < 0 then
    bestbound := limitbound;
  end if;

  if mins >= 0 then
   bests := mins;
   bestbound := subs(s=bests, bound);
  end if;

  eval(bestbound);
end proc;
```

Figure A.4: OpenMaple procedure for Chernoff-Hoeffding bounds.

Then scanning lists $i_1, \ldots, i_k$ to depths $w_{i_1}, \ldots, w_{i_k}$, respectively, yields a total scan depth $\leq C$ (and we can get exactly C by scanning a few more positions without further score decrease in any of the lists) and a total score decrease of $u_{i_1} + ... + u_{i_k} \geq U$.
Proof of $(B) \Rightarrow (A)$:
Given (B), let $i_1, \ldots, , i_k$ be the lists where a non-zero score decrease has been achieved. List i_j has then been scanned at least to depth w_{i_j}, and therefore the total scan depth C is at least $w_{i_1} + ... + w_{i_k}$. The total score reduction of these lists is exactly $u_{i_1} + ... + u_{i_k}$, which by (B) is $\geq U$. The choice of items $i_1, \ldots, i_k$ yields a packing that satisfies (A).

A.3.2 Lower Bound for the Index Access Scheduling Problem

Fagin et al. [FLN03] proved that their CA algorithm has costs that are always within a (constant) factor of $4m + k$ of the optimum for an given query, data set, and monotonous score aggregation function. Recall that m is the number

of lists and k is the number of top items we want to see. Note that even for relatively small values of m and k, the above bound may become fairly large (e.g., 22 for a typical IR-style keyword query with $k = 10$ top results required and $m = 3$ keywords), and, it seems, way too pessimistic.

We instead compute a lower bound on the cost of *individual* queries, and compare these to the costs of our various schemes. The key idea of this lower bound is as follows. For any scheme, after it has done its last sorted access, consider the set X of documents which were seen in one at least of the sorted accesses, and which have a bestscore not only above the current *min-k* score, but even above the final *min-k* score (which the scheme does not know at this time). If only a fraction of each list has been scanned, this set X is typically of considerable size. Now it is not hard to see that the scheme *must do a random lookup for every document from* X in order to be correct. (Otherwise, let d be one of the documents that are not looked up, and consider input lists, where d comes right after where our scheme has stopped scanning the lists, achieving the maximal score still possible then. Then d is one of the top-k items, but our scheme will fail to recognize it as such.)

Therefore, the following construction gives a lower bound on the cost of *any* top-k scheme, under the reasonable assumption that random lookups are done only for documents which have been previously seen under sorted access [FLN03]: try all possible combinations of scan depths in each of the input lists, and for each such combination compute the cost of scanning until this depth plus the cost of the then absolutely necessary random accesses according to the explanation above. Trying out indeed *all* combinations of possible scan depths is, of course, infeasible, but we can restrict ourselves to scan depths that are multiples of a certain block size. This will give us a true lower bound for any scheme that indeed proceeds in blocks of this fixed size (and we consider only such schemes in this paper). But even for an arbitrary scheme, the optimal cost could be better by at most this block's size times the number of input lists.

A.4 Customized Queries

A.4.1 IMDB Relational Queries

<num> IMDB1
<title> Genre=Western Actor=Wayne_John Actor=Hepburn_Katherine Sheriff Marshall

<num> IMDB2
<title> Genre=Western Actor=Fonda_Henry Outlaw

<num> IMDB3
<title> Genre=Western Actor=Newman_Paul Title=Outrage

<num> IMDB4
<title> Genre=Western Actor=Wayne_John Indians Title=Dorado

<num> IMDB5
<title> Genre=Action Actor=Reeves_Keanu Martial Arts Fight Title=Matrix

<num> IMDB6
<title> Genre=Thriller Actor=Pitt_Brad Actor=Freeman_Morgan Murder Title=Seven

<num> IMDB7
<title> Genre=Thriller Actor=Schwarzenegger_Arnold Robot

<num> IMDB8
<title> Genre=Comedy Actor=Allen_Woody Woman

<num> IMDB9
<title> Genre=Comedy Tom Hanks Vietnam War Title=Gump

<num> IMDB10
<title> Genre=SciFi Actor=Roberts_Julia Alien Space

<num> IMDB11
<title> Genre=SciFi Actor=Ford_Harrison Robot War Title=Space

<num> IMDB12
<title> Genre=Film-Noir Genre=Thriller Actor=Marlowe_Frank Chicago Prohibition

<num> IMDB13
<title> Genre=Drama Actor=Ozari_Romano Title=Nosferatu

<num> IMDB14
<title> Genre=Drama Actor=Seymour_Dan World War

<num> IMDB15
<title> Genre=Thriller Actor=Bogart_Humphrey War Title=Casablanca

<num> IMDB16
<title> Actor=Welles_Orson Rosebud

<num> IMDB17
<title> Genre=Thriller Title=3rd Title=Man

<num> IMDB18
<title> Genre=Horror Actor=Lee_Christopher Coffin Blood Vampire

<num> IMDB19
<title> Genre=Crime Actor=Sims_Joan Marple Paddington

<num> IMDB20
<title> Genre=Action Actor=Dalton_Timothy SPECTRE Title=007

A.4.2 IMDB NEXI Queries

```
<num>     X-IMDB1
<title>   //movie[about(.//keyword, bank robbery) and about(.//keyword, crime)
              and about( .//genres//genre, Drama)]
<num>     X-IMDB2
<title>   //movie[about(.//keyword, mafia) and about( .//cast//actor//birthplace, Italy)
              and about(.//genre, Action)]
<num>     X-IMDB3
<title>   //movie[about(.//keyword, road movie) and about(.//cast//birthplace, Canada)]
<num>     X-IMDB4
<title>   //movie[about(.//genres//genre, Action) and about(//title, sea)]
              and //cast[about( .//casting//actor//birthplace, California)]
<num>     X-IMDB5
<title>   //movie[about(//plot, monster)] and //cast[about(.//actor//birthplace, Australia)]
              and //genres[about(.//genre, Horror)]
<num>     X-IMDB6
<title>   //movie[about(.//title, Killer) and about(.//genres//genre, Thriller)]
<num>     X-IMDB7
<title>   //movie[about(.//cast//casting//actor//birthplace, England)
              and about(.//plot, vampire) and about(.//genres//genre, Horror)]
<num>     X-IMDB8
<title>   //movie[about(., revolution)] and //cast[about(.//actor//birthplace, Africa)]
<num>     X-IMDB9
<title>   //movie[about(.//title, love) and about(.//cast//casting//actor//birthplace, India)
              and about(.//genre, Comedy)]
<num>     X-IMDB10
<title>   //movie[about(.//cast//actor//name, John Wayne)]
              and about(.//cast//actor//name, Kirk Douglas)]
<num>     X-IMDB11
<title>   //movie[about(.//cast//casting//role, Sheriff)
              and //casting//actor[about(.//name, Fonda Henry)]]
<num>     X-IMDB12
<title>   //movie[about(.//genres//genre, Sci-Fi)]//cast//casting//[about(.//role, Neo)
              and about(.//actor//name, Keanu Reeves)]
              and //movie[about(.//genre, Action)]
<num>     X-IMDB13
<title>   //movie[about(.//actor//name, Stan Laurel) and about(.//actor//name,
              Oliver Hardy) and about(.//movie, Hollywood) and about(.//genre, Comedy)]
<num>     X-IMDB14
<title>   //movie[about(.//actor//name, Arnold Schwarzenegger)
              and about(.//movie//keyword, nuclear war) and about(.//genre, Action)]
<num>     X-IMDB15
<title>   //movie[about(.//actor//name, Woody Allen) and about(.//actor//birthplace,
               England) and about(.//genre, Drama)]
<num>     X-IMDB16
<title>   //movie[about(.//plot, Dracula) and about(.//cast//casting//actor//name, Lee)
              and about(.//genre, Horror)]
<num>     X-IMDB17
<title>   //movie[about(.//title, Star Trek) and about(.//movie//plot, planet)]
              and //cast[about( .//casting//actor//birthplace, USA)]
<num>     X-IMDB18
<title>   //movie[about(.//title, Matrix) and about(.//cast//casting//actor//name, Reeves)]
<num>     X-IMDB19
<title>   //movie[about(.//movie//plot, Enterprise Spock Planet)
              and about(.//plot, Captain Kirk) and about(.//genre, Sci-Fi)]
<num>     X-IMDB20
<title>   //movie[about(.//cast//actor//name, Weaver) and about(.//plot, alien planet)
              and about(.//genre, Sci-Fi)]
```

A.4.3 Extended GOV (XGOV) Queries

<num> TDEX1

<title> <expansion>mining gold silver coal metal location mineral resources industry

<desc> What can be learned about the location of mines in the U.S., about the extent of mineral resources, and about careers in the mining industry?

<num> TDEX2

<title> juvenile delinquency youth minor crime law jurisdiction offense prevention

<desc> What are rates of juvenile crime in various jurisdictions, what is the nature of the offenses, how are they punished and what measures are taken for prevention?

<num> TDEX3

<title> Lewis and Clark expedition historic explore

<desc> What are some useful sites containing information about the historic Lewis and Clark expedition?

<num> TDEX4

<title> wireless communications radio broadcasting transmission electromagnetic waves use research technology regulations legislative

<desc> Information on existing and planned uses, research/technology, regulations and legislative interest.

<num> TDEX5

<title> pest control safety epidemic contamination quarantine

<desc> Where can I obtain information about safe means of pest control?

<num> TDEX6

<title> physical therapists healer training licensing skills body

<desc> How can I obtain information about training, licensing, and skills needed for physical therapists?

<num> TDEX7

<title> cotton industry growing harvesting cloth silky fiber plant fabric textile material

<num> TDEX8

<title> computer viruses software program malevolent worm trojan bug

<desc> Computer viruses information

<num> TDEX9

<title> genealogy searches family tree lineage bloodline descent ancestry pedigree origin parentage generation

<desc> How would I begin a genealogy search of my family?

<num> TDEX10

<title> Physical Fitness shape condition body training

<desc> Information on Physical Fitness

<num> TDEX11

<title> folk art folk music ethnic traditional song ballad country western gospel singing

<desc> What are sources for information about US folk art and folk music?

<num> TDEX12

<title> legalization marijuana cannabis drug soft leaves plant smoked chewed euphoric abuse substance possession control pot grass dope weed smoke <desc> Where can I locate information on the pros and

cons of legalizing marijuana?

<num> TDEX13

<title> schizophrenia disorder psychosis distortion reality disturbance social contact

<desc> What is it?

<num> TDEX14

<title> agricultural biotechnology farming cultivation land food grow crops microorganism bacteria industrial process genetically altered

<desc> Information about agricultural biotechnology.

<num> TDEX15

<title> cell phones cellular mobile hand-held radio transmitter receiver wireless telephone electronic signal sound

<desc> What does the government say about cell phone use?

<num> TDEX16

<title> Emergency disaster preparedness assistance local state national crisis danger immediate action catastrophe extreme readiness help aid

<desc> Find information about local, state and national organizations and programs.

<num> TDEX17

<title> Polygraphs requirement exam medical instrument physiological process pulse rate blood pressure respiration perspiration lie detector

<desc> Need information on polygraphs and polygraph exams including the requirement to take such exams.

<num> TDEX18

<title> shipwrecks ship wreck accident sea capsizing boat nautical water

<desc> Where can I get information on shipwrecks?

<num> TDEX19

<title> cybercrime internet fraud cyber detection crime

<desc> Information on cyber crime, internet fraud, and cyber fraud.

<num> TDEX20

<title> children's literature youngster kid book writing novel

<desc> What can I find out about literature written for children?

<num> TDEX21

<title> cartography mapmaking map chart

<desc> Cartography overview.

<num> TDEX22

<title> veteran's benefits ex-serviceman financial assistance

<desc> Seeking information on veteran's benefits.

<num> TDEX23

<title> photography picture taking telephotography

<desc> I need information on photography.

<num> TDEX24

<title> airbag air bag safety restraint automobile inflate collision

<desc> Information on the success of air bags in reducing injuries/death; facts concerning the use of air bags and how they work.
<num> TDEX25
<title> death penalty execution executing capital punishment hanging electric chair electrocution arguments lawyers
<desc> What are the costs and quality of defense in death penalty cases? What are some arguments against the death penalty?
<num> TDEX26
<title> nuclear atomic power plants power station power house
<desc> Operational and safety information associated with nuclear power plants.
<num> TDEX27
<title> affirmative action discrimination minority groups
<desc> What are the laws/regulations/policies guiding affirmative action for federal agencies?
<num> TDEX28
<title> early childhood child infancy babyhood education instruction teaching pedagogy
<desc> Looking for pages regarding resources, research, Head Start and similar programs.
<num> TDEX29
<title> asbestos fibrous amphibole asbestosis
<desc> Facts about asbestos.
<num> TDEX30
<title> counterfeit imitation forgery fake false forged money paper coin
<desc> What sites can one go to to learn about counterfeit money and forgery?
<num> TDEX31
<title> deafness deaf hearing loss deaf-mutism deaf-muteness children child kids youngsters preschooler infant baby anxiety disorder
<desc> Childhood deafness, education, communication, audiology. What can we learn about the education or communication of deaf children?
<num> TDEX32
<title> wildlife living undomesticated conservation preservation conservancy environment
<desc> Wildlife conservation-environment protection-endangered species (How do government agencies address wildlife conservation, and strive to further environmental protection and benefit endangered species?)
<num> TDEX33
<title> food nutrient foodstuff comestible edible eatable eat safety risklessness security
<desc> Food safety (what information is available for consumers interested in food safety?)
<num> TDEX34
<title> literacy center ability read write human skills learn knowledge cognition
<desc> What are some pages to search for literacy related topics?
<num> TDEX35
<title> arctic north-polar north pole exploration geographical expedition discovery
<desc> What kinds of exploration of the arctic are underway, especially of glaciers and ice?
<num> TDEX36

<title> global warming increase average temperature earth atmosphere climatic changes planetary worldwide heating
<desc> What are some causes of global warming or the greenhouse effect and what remedies are proposed to slow the climate change?
<num> TDEX37
<title> coin collecting numismatics numismatology coin collection
<desc> Coin collecting information?
<num> TDEX38
<title> weather hazards and extremes peril risk jeopardy wind rain snow storm wave
<desc> A study of natural/weather hazards and extremes.
<num> TDEX39
<title> National Public Radio/TV television telecasting broadcasting cable
<desc> What sites give me information about public radio and TV stations?
<num> TDEX40
<title> north korea democratic people's republic of korea DPRK communist country
<desc> Where can I find basic information about the country of North Korea?
<num> TDEX41
<title> electric automobiles production car research progress fuel
<desc> Need information regarding the progress in producing/developing electric automobiles.
<num> TDEX42
<title> homelessness combat vagrancy wandering livelihood home prevalence
<desc> What is the prevalence of homelessness and how are government agencies and individuals attempting to combat it?
<num> TDEX43
<title> forest fires woods burn flames dry summer
<desc> Where can I get information about forest fires?
<num> TDEX44
<title> ozone layer environment pollution ultraviolet rays industry
<desc> Interested in any information on the ozone layer.
<num> TDEX45
<title> bicycle trails mountain bike downhill sport offroad nature
<desc> Where can I find trails that bicyclists can exploit?
<num> TDEX46
<title> infant mortality death rate children neonatal
<desc> What are the current trends in infant mortality and what steps have been taken to reduce infant death rates?
<num> TDEX47
<title> trains railroads travel safety government industry
<desc> What information is available on train travel, safety, and industry government support?
<num> TDEX48
<title> robots artificial machine production lane research

<desc> What home pages will bring me information on the use of robots?

<num> TDEX49

<title> bilingual education language learning skills school children

<desc> Looking for home pages on bilingual education.

<num> TDEX50

<title> anthrax bacillus anthracis fever disease treatment prevention contagion quarantine

<desc> Info regarding prevention and treatment of the disease anthrax.

```
uniformbound := proc(delta, higharray, n);
  totalhigh := 0;
  for i from 1 to n do
    totalhigh := totalhigh + higharray[i];
  end do;
  boundarray := array(1..n);
  for i from 1 to n do
    boundarray[i] := 0;
  end do;

  chernoffbound := proc (d, h);
    f1(x) := 1 / h;
    L1 := int (exp(-s * x) * f1(x), x=0..h);
    delta1 := d;
    bound := exp(-s * delta1) * subs(s=-s, L1);
    limitbound := limit(bound, s=0, right);
    diffbound := diff(bound, s);
    eq := diffbound = 0;
    mins := fsolve(eq, s);
    if mins < 0 then
      bestbound := limitbound;
    end if;
    if mins >= 0 then
      bests := mins;
      bestbound := subs(s=bests, bound);
    end if;
    evalf(bestbound);
  end proc;

  for i from 1 to n do
    if higharray[i] > 0 then
      boundarray[i] :=
        chernoffbound(
          evalf(delta * higharray[i] / totalhigh),
          evalf(higharray[i])
        );
    else
      boundarray[i] := 0;
    end if;
  end do;

  maxbound := 0;
  for i from 1 to n do
    maxbound := evalf(max(boundarray[i], maxbound));
  end do;
end proc;
```

Figure A.5: OpenMaple procedure for generalized Chernoff-Hoeffding bounds.

Bibliography

[AAC+01] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, pages 591–600, 2001.

[ACDG03] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[ACR03] G. Amati, C. Carpineto, and G. Romano. Fondazione Ugo Bordoni at TREC 2003: Robust and Web Track. In *TREC 2003*, pages 234–245, 2003.

[AdKM01] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In Croft et al. [CHKZ01], pages 35–42.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *PDIS*, pages 208–219. IEEE Computer Society, 1996.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 261–272. ACM, 2000.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A

primitive for efficient XML query pattern matching. In *ICDE* [DBL02], pages 141–.

[AKYJ03] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying structured text in an XML database. In Halevy et al. [HID03], pages 4–15.

[All90] Arnold O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications, 2nd ed.* Academic Press, 1990.

[ARSZ03] Giuseppe Amato, Fausto Rabitti, Pasquale Savino, and Pavel Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, 2003.

[AvR02] Gianni Amati and C. J. van Rijsbergen. Probabilistic models of Information Retrieval based on measuring the divergence from randomness. *ACM Trans. Inf. Syst.*, 20(4):357–389, 2002.

[AYBS04] Sihem Amer-Yahia, Chavdar Botev, and Jayavel Shanmugasundaram. TeXQuery: a full-text search extension to XQuery. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW*, pages 583–594. ACM, 2004.

[AYFSX03] Sihem Amer-Yahia, Mary F. Fernández, Divesh Srivastava, and Yu Xu. Pix: A system for phrase matching in XML documents. In Dayal et al. [DRV03], pages 768–776.

[AYLP04] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible structure and full-text querying for XML. In Weikum et al. [WKD04], pages 83–94.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[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.

[BCG02] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[BGM02] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over Web-accessible databases. In *ICDE* [DBL02], pages 369–.

[BJH+05] Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005.* ACM, 2005.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In Franklin et al. [FMA02], pages 310–321.

[BL85] Chris Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.

[BMS+06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. IO-Top-k: Index-access optimized top-k query processing. In *Technical Report MPI-I-2006-5-002*, 2006.

[BMT+06] Holger Bast, Debapriyo Majumdar, Martin Theobald, Ralf Schenkel, and Gerhard Weikum. IO-Top-k: Index-optimized top-k query processing. In *VLDB*, 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.

[BSA94] Chris Buckley, Gerard Salton, and James Allan. The effect of adding relevance information in a relevance feedback environment. In Croft and van Rijsbergen [CvR94], pages 292–300.

[BSWZ03] Bodo Billerbeck, Falk Scholer, Hugh E. Williams, and Justin Zobel. Query expansion using associated queries. In *CIKM* [DBL03], pages 2–9.

[BV00] Chris Buckley and Ellen M. Voorhees. Evaluating evaluation measure stability. In *SIGIR*, pages 33–40, 2000.

[BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[BYZM+05] Ricardo A. Baeza-Yates, Nivio Ziviani, Gary Marchionini, Alistair Moffat, and John Tait, editors. *SIGIR 2005: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Salvador, Brazil, August 15-19, 2005*. ACM, 2005.

[BZ04a] Bodo Billerbeck and Justin Zobel. Questioning query expansion: An examination of behaviour and parameters. In Klaus-Dieter Schewe and Hugh E. Williams, editors, *ADC*, volume 27 of *CRPIT*, pages 69–76. Australian Computer Society, 2004.

[BZ04b] Bodo Billerbeck and Justin Zobel. Techniques for efficient query expansion. In Alberto Apostolico and Massimo Melucci, editors, *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2004.

[CCS04] Charles L. A. Clarke, Nick Craswell, and Ian Soboroff. Overview of the TREC 2004 Terabyte track. In *TREC*, pages 78–92, 2004.

[CCS05] Charles L. A. Clarke, Nick Craswell, and Ian Soboroff. The TREC Terabyte retrieval track. *SIGIR Forum*, 39(1):25, 2005.

[CGM04] Surajit Chaudhuri, Luis Gravano, and Amélie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[CH04] Nick Craswell and David Hawking. Overview of the TREC 2004 Web track. In *TREC*, pages 78–92, 2004.

[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.

[CHWW03] Nick Craswell, David Hawking, Ross Wilkinson, and Mingfang Wu. Overview of the TREC 2003 Web track. In *TREC*, pages 78–92, 2003.

[CK97] Michael J. Carey and Donald Kossmann. On saying "enough already!" in sql. In Joan Peckham, editor, *SIGMOD Conference*, pages 219–230. ACM Press, 1997.

[CK01] Taurai Tapiwa Chinenyanga and Nicholas Kushmerick. Expressive retrieval from XML documents. In Croft et al. [CHKZ01], pages 163–171.

[Cla05] Charles L. A. Clarke. Controlling overlap in content-oriented XML retrieval. In Baeza-Yates et al. [BYZM+05], pages 314–321.

[CLRC01] Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest, and Stein Clifford. *Introduction of Algorithms*. The MIT Press, 2001.

[CMKS03] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: A semantic search engine for XML. In Freytag et al. [FLA+03], pages 45–56.

[CMM+03] David Carmel, Yoëlle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching XML documents via XML fragments. In *SIGIR*, pages 151–158. ACM, 2003.

[CMW03] Byron Choi, Malika Mahoui, and Derick Wood. On the optimality of holistic algorithms for twig queries. In Vladimír Marík, Werner Retschitzegger, and Olga Stepánková, editors, *DEXA*, volume 2736 of *Lecture Notes in Computer Science*, pages 28–37. Springer, 2003.

[CP00] Paolo Ciaccia and Marco Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *ICDE*, pages 244–255, 2000.

[CRZT05] Nick Craswell, Stephen E. Robertson, Hugo Zaragoza, and Michael Taylor. Relevance weighting for query independent evidence. In Baeza-Yates et al. [BYZM+05], pages 416–423.

[CSF+01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In Apers et al. [AAC+01], pages 341–350.

[CTZC04] Stephen Cronen-Townsend, Yun Zhou, and W. Bruce Croft. A framework for selective query expansion. In Grossman et al. [GGZ+04], pages 236–237.

[CvR94] W. Bruce Croft and C. J. van Rijsbergen, editors. *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum).* ACM/Springer, 1994.

[CwH02] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In Franklin et al. [FMA02], pages 346–357.

[DBL02] *ICDE 2002: Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[DBL03] *CIKM 2003: Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*. ACM, 2003.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In Nascimento et al. [NÖK+04], pages 948–959.

[DR99] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB*, pages 411–422. Morgan Kaufmann, 1999.

[DRV03] Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[dVMNK02] Arjen P. de Vries, Nikos Mamoulis, Niels Nes, and Martin L. Kersten. Efficient k-nn search on vertically decomposed data. In Franklin et al. [FMA02], pages 322–333.

[EM03] Nadav Eiron and Kevin S. McCurley. Analysis of anchor text for Web search. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 459–460, New York, NY, USA, 2003. ACM Press.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[Feg04] Leonidas Fegaras. XQuery processing with relevance ranking. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2004.

[Fel98] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[FG01] Norbert Fuhr and Kai Großjohann. XIRQL: A query language for Information Retrieval in XML documents. In Croft et al. [CHKZ01], pages 172–180.

[FLA+03] Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors. *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*. Morgan Kaufmann, 2003.

[FLMS05] Norbert Fuhr, Mounia Lalmas, Saadia Malik, and Zoltán Szlávik, editors. *Advances in XML Information Retrieval, Third International Workshop of the INitiative for the Evaluation of XML Retrieval, INEX 2004, Dagstuhl Castle, Germany, December 6-8, 2004, Revised Selected Papers*, volume 3493 of *Lecture Notes in Computer Science*. Springer, 2005.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*. ACM, 2001.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[FMA02] Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. ACM, 2002.

[FZ05] Hui Fang and ChengXiang Zhai. An exploration of axiomatic approaches to Information Retrieval. In Baeza-Yates et al. [BYZM+05], pages 480–487.

[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*, 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.

[GF05] D. A. Grossman and O. Frieder. *Information Retrieval*. Springer Verlag, 2005.

[GGZ+04] David Grossman, Luis Gravano, ChengXiang Zhai, Otthein Herzog, and David A. Evans, editors. *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004*. ACM, 2004.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190. ACM, 2003.

[Gru02] Torsten Grust. Accelerating XPath location steps. In Franklin et al. [FMA02], pages 109–120.

[GS03] Torsten Grabs and Hans-Jörg Schek. PowerDB-XML: Scalable XML processing with a database cluster. In *Intelligent Search on XML Data*, pages 193–206, 2003.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRank: Ranked keyword search over XML documents. In Halevy et al. [HID03], pages 16–27.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 47–57. ACM Press, 1984.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB*, pages 436–445. Morgan Kaufmann, 1997.

[HAR75] S. P. HARTER. A probabilistic approach to automatic keyword indexing. part 1: "on the distribution of speciality words in a technical literature", part 2: "an algorithm for probabilistic indexing". *Journal of the American Society for Information Science*, 26:197–206 and 280–289, 1975.

[HCO03] Chien-Kang Huang, Lee-Feng Chien, and Yen-Jen Oyang. Relevant term suggestion in interactive Web search based on contextual information in query session logs. *JASIST*, 54(7):638–649, 2003.

[HDS04] Edward Hung, Yu Deng, and V. S. Subrahmanian. TOSS: An extension of TAX with ontologies and similarity queries. In Weikum et al. [WKD04], pages 719–730.

[HID03] Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[HPB03] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on XML graphs. In Dayal et al. [DRV03], pages 367–378.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[HS03] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[HT95] David Hawking and Paul B. Thistlewaite. Proximity operators - so near and yet so far. In *TREC*, 1995.

[IAE03] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[IAE04] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[IDF] The IDF page. `http://www.soi.city.ac.uk/~ser/idf.html`.

[INE] INitiative for the Evaluation of XML Retrieval (INEX). `http://inex.is.informatik.uni-duisburg.de`.

[Ioa03] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[IOT] Index-Organized Tables – Oracle9*i* Data Sheet. `http://www.oracle.com/technology/products/oracle9i/datasheets/iots/iot\%_ds.html`.

[ISA+04] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In Weikum et al. [WKD04], pages 203–214.

[ITA] The Internet Traffic Archive. `http://ita.ee.lbl.gov/`.

[JLST01] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In Giorgio Ghelli and Gösta Grahne, editors, *DBPL*, volume 2397 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2001.

[JWLY03] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In Freytag et al. [FLA+03], pages 273–284.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In Bharat K. Bhargava, Timothy W. Finin, and Yelena Yesha, editors, *CIKM*, pages 490–499. ACM, 1993.

[KG90] M. Kenall and J.D. Gibbons. *Rank Correlation Methods.* Oxford University Press, 1990.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In Weikum et al. [WKD04], pages 779–790.

[KL05] Gabriella Kazai and Mounia Lalmas. INEX 2005 evaluation metrics. In *INEX*, 2005.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[Kwo04] Kui-Lam Kwok. TREC 2004 Robust track experiments using PIRCS. In *TREC*, 2004.

[LC01] Victor Lavrenko and W. Bruce Croft. Relevance-based language models. In Croft et al. [CHKZ01], pages 120–127.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142, 2005.

[Lid20] G. J. Lidstone. Note on the general case of the bayes-laplace formula for inductive or a posteriori probabilities. In *Transactions of the Faculty of Actuaries*, volume 8, pages 182–192, 1920.

[LLYM04] Shuang Liu, Fang Liu, Clement Yu, and Weiyi Meng. An effective approach to document retrieval via utilizing WordNet and recognizing phrases. In Mark Sanderson, Kalervo Järvelin, James Allan, and Peter Bruza, editors, *SIGIR*, pages 266–272. ACM, 2004.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Apers et al. [AAC+01], pages 361–370.

[LS03] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In Freytag et al. [FLA+03], pages 129–140.

[LSCI05] Chengkai Li, Mohamed A. Soliman, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Ranksql: Supporting ranking queries in relational database management systems. In Böhm et al. [BJH+05], pages 1342–1345.

[LWP+02] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ronald Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In *VLDB*, pages 442–453, 2002.

[LYJ04] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free XQuery. In Nascimento et al. [NÖK+04], pages 72–83.

[MAYKS05] Amélie Marian, Sihem Amer-Yahia, Nick Koudas, and Divesh Srivastava. Adaptive processing of top-k queries in XML. In *ICDE*, pages 162–173. IEEE Computer Society, 2005.

[MBG04] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over Web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[MS99a] Christopher D. Manning and Hinrich Sch"utze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[MS99b] Tova Milo and Dan Suciu. Index structures for path expressions. In Catriel Beeri and Peter Buneman, editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.

[MSB98] Mandar Mitra, Amit Singhal, and Chris Buckley. Improving automatic query expansion. In *SIGIR*, pages 206–214. ACM, 1998.

[MTV04] D. Mavroeidis, G. Tsatsaronis, and M. Vazirgiannis. Semantic distances for sets of senses and applications in word sense disambiguation. In *Proceedings of the Knowledge Mining NEMIS 2004 Final Conference*, 2004.

[MTV+05] Dimitrios Mavroeidis, George Tsatsaronis, Michalis Vazirgiannis, Martin Theobald, and Gerhard Weikum. Word sense disambiguation for exploiting hierarchical thesauri in text classification. In Alípio Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *PKDD*, volume 3721 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2005.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[NCS+01] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In Apers et al. [AAC+01], pages 281–290.

[Nel95] Randolph Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[NO00] Masumi Narita and Yasushi Ogawa. The use of phrases from query texts in Information Retrieval. In *SIGIR*, pages 318–320, 2000.

[NÖK+04] Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 2004.

[NR99] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29. IEEE Computer Society, 1999.

[OPE] OpenCyc. `http://www.opencyc.org/`.

[PF95] Ulrich Pfeifer and Norbert Fuhr. Efficient processing of vague queries using a data stream approach. In Edward A. Fox, Peter Ingwersen, and Raya Fidel, editors, *SIGIR*, pages 189–197. ACM Press, 1995.

[PFTV92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1992.

[PGI04] Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate XML query answers. In Weikum et al. [WKD04], pages 263–274.

[QF93] Yonggang Qiu and Hans-Peter Frei. Concept based query expansion. In Robert Korfhage, Edie M. Rasmussen, and Peter Willett, editors, *SIGIR*, pages 160–169. ACM, 1993.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In Dayal et al. [DRV03], pages 353–.

[RJ76] Stephen Robertson and Karen Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(1):129–146, 1976.

[Rob81] Stephen E. Robertson. Term frequency and term value. In Carolyn J. Crouch, editor, *SIGIR*, pages 22–29. ACM, 1981.

[Roc71] J.J. Rocchio Jr. Relevance feedback in Information Retrieval. In G. Salton, editor, *The SMART Retrieval System: Experiments in Automatic Document Processing*, chapter 14, pages 313–323. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1971.

[RW94] Stephen E. Robertson and Steve Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In Croft and van Rijsbergen [CvR94], pages 232–241.

[RW97] Stephen E. Robertson and Steve Walker. On relevance weights with little relevance information. In *SIGIR*, pages 16–24. ACM, 1997.

[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 Taylor. Simple BM25 extension to multiple weighted fields. In Grossman et al. [GGZ+04], pages 42–49.

[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.

[SM02] Torsten Schlieder and Holger Meuss. Querying and ranking XML documents. *JASIST*, 53(6):489–503, 2002.

[SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995.

[SSTW02] Sergej Sizov, Stefan Siersdorfer, Martin Theobald, and Gerhard Weikum. The BINGO! focused crawler: From bookmarks to archetypes. In *ICDE* [DBL02], pages 337–338.

[ST06a] Ralf Schenkel and Martin Theobald. Feedback-driven structural query expansion for ranked retrieval of XML data. In *to be published in EDBT*, 2006.

[ST06b] Ralf Schenkel and Martin Theobald. Structural feedback for keyword-based XML retrieval. In *to be published in ECIR*, 2006.

[STSW02] Sergej Sizov, Martin Theobald, Stefan Siersdorfer, and Gerhard Weikum. BINGO!: Bookmark-induced gathering of information. In Tok Wang Ling, Umeshwar Dayal, Elisa Bertino, Wee Keong Ng, and Angela Goh, editors, *WISE*, pages 323–332. IEEE Computer Society, 2002.

[SWK+02] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.

[TFP03] Yufei Tao, Christos Faloutsos, and Dimitris Papadias. The Power-method: a comprehensive estimation technique for multi-dimensional queries. In *CIKM* [DBL03], pages 83–90.

[TRE] Text REtrieval Conference (TREC). `http://trec.nist.gov/`.

[TS04a] Andrew Trotman and Börkur Sigurbjörnsson. Narrowed Extended XPath I (NEXI). In Fuhr et al. [FLMS05], pages 16–40.

[TS04b] Andrew Trotman and Börkur Sigurbjörnsson. NEXI, Now and Next. In Fuhr et al. [FLMS05], pages 41–53.

[TS05] Martin Theobald and Ralf Schenkel. TopX & XXL @ INEX 2005. In *to be published in INEX 2005*, 2005.

[TSW03] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Exploiting structure, annotation, and ontological knowledge for automatic classification of XML data. In Vassilis Christophides and Juliana Freire, editors, *WebDB*, pages 1–6, 2003.

[TSW05a] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In Baeza-Yates et al. [BYZM+05], pages 242–249.

[TSW05b] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for TopX search. In Böhm et al. [BJH+05], pages 625–636.

[TW00] Anja Theobald and Gerhard Weikum. Adding relevance to XML. In Dan Suciu and Gottfried Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2000.

[TW02] Anja Theobald and Gerhard Weikum. The index-based XXL Search Engine for querying XML data with relevance ranking. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *EDBT*, volume 2287 of *Lecture Notes in Computer Science*, pages 477–495. Springer, 2002.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, pages 130–141. ACM Press, 1998.

[VMT04] Zografoula Vagena, Mirella Moura Moro, and Vassilis J. Tsotras. Twig query processing over graph-structured XML data. In Sihem Amer-Yahia and Luis Gravano, editors, *WebDB*, pages 43–48, 2004.

[Voo94] Ellen M. Voorhees. Query expansion using lexical-semantic relations. In Croft and van Rijsbergen [CvR94], pages 61–69.

[Vor04] Ellen Vorhees. Overview of the TREC 2004 Robust retrieval track. In *TREC*, pages 69–77, 2004.

[W3Ca] XQuery 1.0 and XPath 2.0 Full-Text. `http://www.w3.org/TR/xquery-full-text/`.

[W3Cb] Extensible Markup Language (XML). http://www.w3.org/XML/.

[W3Cc] XML Path Language (XPath). http://www.w3.org/TR/xpath/.

[W3Cd] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[WIK] Wikipedia, the free encyclopedia. http://www.wikipedia.org/.

[WKD04] Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.

[WPJ03] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In Dayal et al. [DRV03], pages 443–454.

[XC96] Jinxi Xu and W. Bruce Croft. Query expansion using local and global document analysis. In Hans-Peter Frei, Donna Harman, Peter Schäuble, and Ross Wilkinson, editors, *SIGIR*, pages 4–11. ACM, 1996.

[XPO] XML Pointer Language (XPointer). http://www.w3.org/TR/xptr/.

[YSMQ01] Clement T. Yu, Prasoon Sharma, Weiyi Meng, and Yan Qin. Database selection for processing k nearest neighbors queries in distributed environments. In *JCDL*, pages 215–222. ACM, 2001.

[ZL04] ChengXiang Zhai and John D. Lafferty. A study of smoothing methods for language models applied to Information Retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.