University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Open Access Dissertations

9-2013

Query-Time Optimization Techniques for Structured Queries in Information Retrieval

Marc-Allen Cartright
University of Massachusetts Amherst, mcartright@gmail.com

Follow this and additional works at: https://scholarworks.umass.edu/open_access_dissertations

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
Cartright, Marc-Allen, "Query-Time Optimization Techniques for Structured Queries in Information Retrieval" (2013). *Open Access Dissertations*. 779.
https://scholarworks.umass.edu/open_access_dissertations/779

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

QUERY-TIME OPTIMIZATION TECHNIQUES FOR STRUCTURED QUERIES IN INFORMATION RETRIEVAL

A Dissertation Presented

by

MARC-ALLEN CARTRIGHT

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2013

School of Computer Science

© Copyright by Marc-Allen Cartright 2013

All Rights Reserved

QUERY-TIME OPTIMIZATION TECHNIQUES FOR STRUCTURED QUERIES IN INFORMATION RETRIEVAL

A Dissertation Presented

by

MARC-ALLEN CARTRIGHT

Approved as to style and content by:

____
James Allan, Chair

____
W. Bruce Croft, Member

____
David Smith, Member

____
Michael Lavine, Member

____
Howard Turtle, Member

____
Lori A. Clarke, Chair
School of Computer Science

To Ilene, who was there every step of the way.

ACKNOWLEDGMENTS

It's hard to know how much coverage one should give when acknowledging all of the people that helped you get to this point. There is a multitude of people I could thank, all of whom served as teachers or advisers at some point in my life. However the desire to pursue a Ph.D. came relatively late in my life so far, and so to me it makes sense to only mention here the people that helped me come through this experience successfully and mentally intact. Usually James admonishes me for using "flowery language", and so I usually try to tone it down. However, not this time.

I'd like to start by thanking the Center for Intelligent Information Retrieval and the individuals both itinerant and permanent who comprise it. The CIIR provided a home for me academically while I figured out what it meant to be a scientist, and in particular in the discipline of IR. Even after a high-flying internship, it was good to come back to the lab and get back to the environment afforded by it. I'd actually like to thank the lab in two parts: the first are the staff members who keep the whole thing running while we tinker away in our own little worlds, and the second are those tinkerers who provided some of the best conversations I've ever had.

The staff of the CIIR have been an immense help throughout my Ph.D. They kept everything running smoothly and made our lives entirely too comfortable for our own good. In particular, Kate Moruzzi, Jean Joyce, Glenn Stowell, David Fisher,

and Dan Parker have all been amazing, and I can only hope future grad students are as lucky as were to have them.

The other part of the CIIR, the students and scientists in the organization, have made IR one of the most fascinating topics I have ever studied. The environment in the lab has always been one of trying new things and pushing the boundaries of what we think of as search, and I can only hope to be in a similar environment in the future. Our conversations in the lab have been enlightening and sometimes contentious, and I think I'm a better researcher for it. In particular, I'd like to thank Henry Feild, Michael Bendersky, Sam Huston, Niranjan Balasubramanian, Elif Aktolga, Jeff Dalton, Laura Dietz, Van Dang, John Foley, Zeki Yalniz, Ethem Can, Tamsin Maxwell, and Matt Lease. All the best to you in your future endeavors.

Over the course of the six years it took to complete this Ph.D., I have made many friends, all of whom have made this experience that much better. I'm pretty sure the list is longer than I can recall, and I will almost certainly miss people who deserve to be mentioned, but I'm going to list the people I can think of anyhow, because I think deserve it. Note that everyone I mentioned in the CIIR already belong to this group, as my peers in CIIR I also consider my friends outside it. In addition to those individuals, I think Jacqueline Feild, Dirk Ruiken, George Konidaris, Bruno Ribeiro, Scott Kuindersma, Sarah Osentoski, Laura Sevilla Lara, Katerina Marazopoulou, Bobby Simidchieva, Stefan Christov, Gene Novark, Steve and Emily Murtagh, Scott Niekum, Phil Thomas, TJ Brunette, Shiraj Sen, Aruna Balasubramanian, Megan Olsen, Tim Wood, David Cooper, Will Dabney, Karan Hingorani, Jill Graham, Lydia Lamriben and Cameron Carter, are all people who have made my time in graduate

school so much more than just an apprenticeship in science. Thank you all for the great times we spent in grad school. but not *at* grad school. Yes, I have that nagging feeling I missed people. I apologize to those who deserve to be mentioned here, but I failed to remember. Know that I truly meant to add you to this list, and you also deserve my thanks for being part of the trip.

Leeanne Leclerc should also be mentioned among my friends, but she also played the added role of being the Graduate Program Manager through the course of my Ph.D. She juggles dealing with both sitting faculty, and a larger number of people who are training to be faculty, and does a superb job of dealing with both groups. I'm at this point sure that she handled more bureaucracy on my behalf than I'm even aware of, and for that I thank her. I'm terrible at dealing with red tape.

James Allan, my Ph.D. adviser, also deserves immense thanks for his role as both an invaluable adviser, and by the end, a good friend. James exhibited what I think was an inhuman amount of patience with me throughout the process. I often can act like a fire hose - a lot of energy with not a lot of direction. James did a superb job in guiding the energy I had into different projects, which in turn allowed me to try a large number of different topics before honing in on a thesis topic. In retrospect, I think there may have been a large number of times where James told me what to do, without actually ordering me to do it. In other words, James is one of the most diplomatic people I have ever seen, and I've tried my best to learn from, and in some cases, probably borrow from, his playbook when interacting with people. I also came to appreciate his pragmatic and direct style of advising - both for myself, as well as his research group as a whole. Only in talking to Ph.D. students in different

situations did I gain the perspective needed to realize that James is in fact a great adviser. I will indeed miss our meetings, which by the end of the Ph.D., were an amalgam of research, engineering, and discussion about pop culture.

I think Bruce Croft, Ryen White, Alistair Moffat, Justin Zobel, Shane Culpepper, and Mark Sanderson deserve special mention as well. I have interacted with each of these scientists either as a peer or as a mentee, and each of them taught me a different path to developing and succeeding as a scientist and academic. It has been a singularly illuminating experience to work with and learn from each of them.

I would also like to thank my committee members: Bruce Croft, David Smith, Howard Turtle, and Michael Lavine, for their insightful guidance and exceptional feedback throughout this thesis, and for their patience enduring a surprisingly long oral defense.

Orion and Sebastian also deserve a thanks, for all of their patience and understanding during this experience. I know I haven't always been the most pleasant person to be around, particularly when deadlines have been looming, but they've put up with me and have always done their best to keep my spirits up. Now I have time to return the favor.

More than anyone, I would like to thank Ilene Magpiong. I see her as nothing less than my traveling partner throughout my Ph.D.; she came to Amherst with me, and during her time here made a life for herself and grew to be a scientist in her own right. However having her around amplified the enjoyment of the entire experience past what I could've hoped for. Ilene took care of me when I was sick, but more importantly she patiently and quietly took care of me when I was too absorbed in

my work to properly take care of myself. She kept our house in working order, even when she didn't live in it, and put up with all of my gripes about some experiment not working, or having a bug somewhere in the depths of the code I was working on. I can continue praising her for all she's done for me, but honestly it's just too much to mention here. I do know that now this chapter is over, I'm so excited to start the next chapter with her I can't even describe it. And just as she was there for me, I can now be there for her.

And now, the formal acknowledgments:

> This work was supported in part by the Center for Intelligent Information Retrieval, in part by NSF CLUE IIS-0844226 and in part by NSF grant #IIS-0910884, in part by DARPA under contract #HR0011-06-C-0023 and in part by UMass NEAGAP fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

ABSTRACT

QUERY-TIME OPTIMIZATION TECHNIQUES FOR STRUCTURED QUERIES IN INFORMATION RETRIEVAL

SEPTEMBER 2013

MARC-ALLEN CARTRIGHT

B.S., STANFORD UNIVERSITY

M.S., UNIVERSITY OF NEVADA LAS VEGAS

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor James Allan

The use of information retrieval (IR) systems is evolving towards larger, more complicated queries. Both the IR industrial and research communities have generated significant evidence indicating that in order to continue improving retrieval effectiveness, increases in retrieval model complexity may be unavoidable. From an operational perspective, this translates into an increasing computational cost to generate the final ranked list in response to a query. Therefore we encounter an increasing tension in the trade-off between retrieval effectiveness (quality of result list) and efficiency (the speed at which the list is generated). This tension creates

a strong need for optimization techniques to improve the efficiency of ranking with respect to these more complex retrieval models.

This thesis presents three new optimization techniques designed to deal with different aspects of structured queries. The first technique involves manipulation of interpolated subqueries, a common structure found across a large number of retrieval models today. We then develop an alternative scoring formulation to make retrieval models more responsive to dynamic pruning techniques. The last technique is delayed execution, which focuses on the class of queries that utilize term dependencies and term conjunction operations. In each case, we empirically show that these optimizations can significantly improve query processing efficiency without negatively impacting retrieval effectiveness.

Additionally, we implement these optimizations in the context of a new retrieval system known as Julien. As opposed to implementing these techniques as one-off solutions hard-wired to specific retrieval models, we treat each technique as a "behavioral" extension to the original system. This allows us to flexibly stack the modifications to use the optimizations in conjunction, increasing efficiency even further. By focusing on the behaviors of the objects involved in the retrieval process instead of on the details of the retrieval algorithm itself, we can recast these techniques to be applied only when the conditions are appropriate. Finally, the modular design of these components illustrates a system design that allows improvements to be implemented without disturbing the existing retrieval infrastructure.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **v**

ABSTRACT . . . . **x**

LIST OF TABLES . . . . **xvi**

LIST OF FIGURES . . . . **xix**

CHAPTER

1. INTRODUCTION **1**

1.1 Problem: Bigger and Bigger Collections 2

1.1.1 Solutions 5

1.2 Problem: Bigger and Bigger Queries 8

1.2.1 Solutions 10

1.3 Another Look at the Same Problem 13
1.4 Contributions 14
1.5 Outline 16

2. BACKGROUND **18**

2.1 Terminology 19
2.2 A Brief History of IR Optimization 20

2.2.1 Processing Models & Index Organizations 23
2.2.2 Optimization Attributes 25
2.2.3 Optimizations in Information Retrieval 27

2.3 State-of-the-art Dynamic Optimization 32

2.3.1 Algorithmic Dynamic Optimization 33

2.3.1.1 Upper Bound Estimators 34
2.3.1.2 Maxscore 35
2.3.1.3 Weak-AND 37

2.3.2 Dynamic Optimization using Machine Learning 40

2.3.2.1 Cascade Rank Model 40
2.3.2.2 Selective Pruning Strategy 42

2.4 Query Languages in Information Retrieval 43

2.4.1 Commercial Search Engines 44
2.4.2 Open Source Search Engines 45
2.4.3 Important Constructs 49
2.4.4 Other Areas of Optimization 50

2.4.4.1 Compiler Optimization 50
2.4.4.2 Database Optimization 51
2.4.4.3 Ranked Retrieval in Databases 53

3. FLATTENING QUERY STRUCTURE 55

3.1 Interpolated Queries 58
3.2 Study 1: Modifying a Single Interpolated Subquery 60

3.2.1 Experiments 62
3.2.2 Results Analysis 67

3.3 Study 2: Examining the Correlation Between Exposure and Speedup 70

3.3.1 Experimental Setup 70

3.3.2 Results 73

3.4 Study 3: Ordering by Impact over Document Frequency 73
3.5 Conclusion of Analysis 75

4. ALTERNATIVE SCORING REPRESENTATIONS 77

4.1 Alternative Representation 82

4.1.1 Terminology 82
4.1.2 Algebraic Description 84
4.1.3 Operational Description 86

4.2 Field-Based Retrieval Models 88

4.2.1 PRMS 89
4.2.2 BM25F 89
4.2.3 Rewriting PRMS 90
4.2.4 Rewriting BM25F 93

4.3 Experiments 98
4.4 Results 101

4.4.1 Bounds Sensitivity 105

4.5 Current Limitations of ASRs 108

5. STRATEGIC EVALUATION OF MULTI-TERM SCORE COMPONENTS 110

5.1 Deferred Query Components 113

5.1.1 Inter-Term Dependency Analysis 117
5.1.2 Generating Approximations 119
5.1.3 Completing Scoring 122

5.2 Experimental Structure 124

5.2.1 Collection and Metrics 128

5.3 Results 129

6. A BEHAVIORAL VIEW OF QUERY EXECUTION 140

6.1 Executing a Query in Julien 142
6.2 Representing a Query 148
6.3 The Behavioral Approach 151

6.3.1 Theory of Affordances 151
6.3.2 Built-In Behaviors 153

6.4 Example: Generating New Smoothing Statistics 156

6.4.1 Extending Indri 157
6.4.2 Extending Galago 159
6.4.3 Extending Julien 161

6.5 Implementing Multiple Optimizations Concurrently 165

6.5.1 Implementing Query Flattening in Julien 165
6.5.2 Exposing Alternative Scoring Representations in Julien 170
6.5.3 Implementing Delayed Evaluation Julien 173

6.6 The Drawbacks of Julien 178

7. CONCLUSIONS 181

7.1 Relationship to Indri Query Language 182
7.2 Future Work 187

APPENDIX: OPERATORS FROM THE INDRI QUERY LANGUAGE 192

REFERENCES 197

LIST OF TABLES

Table **Page**

1.1 Execution time per query as the active size of a collection grows, from 1 million to 10 million documents. The first 10 million documents and first 100 queries from the TREC 2006 Web Track, Efficiency Task were used. Times are in milliseconds. 5

3.1 Results for the Galago retrieval system (v3.3) over AQUAINT, GOV2, and ClueWeb-B, using 36, 150, and 50 queries, respectively. The number in the RM3 column is the number of score requests (in millions) using the unmodified algorithm. The numbers in the remaining columns are the percent change relative to the the unmodified RM3 model. We calculate this as $(B - A)/A$, where A is RM3 and B is the algorithm in question. The † indicates a change that is *not* statistically significant. 68

3.2 Statistics over 750 queries run over GOV2. Mean times are in seconds. The ✦ indicates statistical significance at $p \leq 0.02$. The Score and Time columns report the percentage of queries that experienced at least a 10% drop in the given measurement. 69

3.3 Wall-clock time results for the 4 configurations scored using Wand over the Aquaint collection. Experiments conducted using the Julien retrieval system. 69

3.4 Comparing list length and weight ordering for the Max-Flat algorithm. 75

4.1 Statistics on the collections used in experiments. 'M' indicates a scale of millions. The last column shows the average number of tokens per field for that collection. The second value in that column is the standard deviation of the distribution of tokens per field. 100

4.2 Relative scoring algorithm performance over the Terabyte06 collection, broken down by query length. Exhaustive times are reported in seconds, while other times are reported as a ratio of the Exhaustive time. All relative times are statistically significantly different from the baseline time, unless noted by italics. 103

4.3 Relative scoring algorithm performance over the OpenLib collection, broken down by query length. Exhaustive times are reported in seconds, while other times are reported as a ratio of the Exhaustive time. All relative times are statistically significantly different from the baseline time, unless noted by italics. 104

4.4 A breakdown of the number of improved (win) and worsened (loss) queries, by collection, scoring model, and pruning algorithm. 104

4.5 Relative improvement of the actual value runs vs. the estimated value runs. Values are calculated as `actual / estimated`, therefore the lower the value, the greater the impact tight bounds has on the configuration. 107

5.1 Example set of documents. 117

5.2 Document and collection upper and lower frequency estimates for synthetic terms in: (a) a positional index, and (b) a document-level index. 122

5.3 Effectiveness of retrieval using 50 judged queries from the 2006 TREC Terabyte manual runs, measured using MAP on depth $k = 1{,}000$ rankings, and using P@10. Score-safe methods are not shown. Bold values indicates statistical significance relative to Sdm-ms. 130

5.4 Mean average time (MAT) to evaluate a query, in seconds; and the ratio between that time and the baseline sdm-ms approach. A total of 1,000 queries were used in connection with the 426 GB GOV2 dataset. Labels ending with a * indicate mechanisms that are not score-safe. All relationships against sdm-ms were significant. 131

5.5 Relative execution times as a ratio of the time taken by the sdm-ms approach, broken down by query length. The numbers in the row labeled sdm-ms are average execution times in seconds across queries with that many stored terms (not counting generated synthetic terms); all other values are ratios relative to those. Lower values indicate faster execution. Numbers in bold represent statistical significance relative to sdm-ms; labels ending with a * indicate mechanisms that are not score-safe. 132

5.6 Mean average time (MAT) to evaluate a query, in seconds. A total of 41 queries were used in connection with the TREC 2004 Robust dataset. 136

7.1 Mapping eligibility of Indri operators for optimization techniques. 183

LIST OF FIGURES

Figure **Page**

1.1 Growth of the largest single collection for a TREC track, by year. The width of the bar indicates how long that collection served as the largest widely used collection. 2

2.1 The standard steps taken to process a query, from input of the raw query to returning scored results. 21

3.1 A weighted disjunctive keyword query represented as a query tree. 58

3.2 A "non-flat" query tree, representing the query `'new york' baseball teams`. 59

3.3 The general form of an interpolated subquery tree. Each subquery node S_i may be a simple unigram, or it may be a more complex query tree root at S_i. 60

3.4 Four cases under investigation, with varying amounts of mutability. Shaded nodes are immutable. 62

3.5 Reducing the depth of a tree with one mutable subquery node. 63

3.6 Completely flattening the query tree. 65

3.7 Comparison of a deep query tree, vs a wide tree with the same number of scoring leaves. To the dynamic optimization algorithms of today, the two offer the same chances for eliding work. 71

3.8 A plot of sample correlation coefficients between the *ratio* and *time* variables. Most queries show a significant negative correlation. 74

4.1 An example of a query graph that cannot be flattened. 77

4.2 The generic idea of reformulating a query to allow for better pruning. Instead of attempting to prune after calculating every S_i (by aggregating over the sub-graph contributions), we rewrite the leaf scoring functions to allow pruning after each scorer calculation. 80

4.3 Different timings for Exhaustive, Maxscore (ms-orig) and Maxscore$_F$. The x-axis is time since the start of evaluation, and the y-axis is percent of the collection left to evaluate. The query evaluated is query #120 from the TREC Terabyte 2006 Efficiency Track. 81

4.4 The error in the UBE overestimating the actual upper bound of the scorer. The graph above is of BM25F (both original and ASR formulation), over the first 200 queries of the Terabyte 2006 Efficiency track, using the GOV2 collection. 108

4.5 The error in the UBE overestimating the actual upper bound of the scorer. The graph above is of PRMS (both original and ASR formulation), over the first 200 queries of the Terabyte 2006 Efficiency track, using the GOV2 collection. 109

5.1 Contents of R_k after evaluating each document from Table 5.1. The grey entry indicates the candidate at rank $k = 5$, which is used for trimming the list when possible. The top k elements are also stored in a second heap R_k of size k, ordered by the third component shown, min_d. 116

5.2 Execution time (in seconds) against retrieval effectiveness at depth $k = 10$ with effectiveness measured using P@10. Judgments used are for the TB06 collection, using 50 judged queries. 137

5.3 Execution time (in seconds) against retrieval effectiveness at depth $k = 1{,}000$ with effectiveness measured using MAP to depth 1,000. Judgments used are for the TB06 collection, using 50 judged queries. 138

5.4 Execution time (in seconds) as query length increases. Only queries of length 5 or greater from the 10K queries of the TREC 2006 Terabyte comparative efficiency task query set were used. 139

6.1 A component diagram of the basic parts of Julien. 142

6.2 A simple query tree. 149

6.3 A simple query tree, with both features and views shown. 150

6.4 A query Q, with operators exposing different behaviors, is passed to the QueryProcessor, which executes the query and produces a result list R of retrievables. 153

6.5 Incorrectly flattening the query. The semantics of the query are changed because the lower summation operations are deleted. 169

6.6 A simple walk to look for `Bypassable` operators, which are marked as inverted triangles. The `bypass` function is graphically shown below step (b): a triangle can be replaced by two square operators, which when summed produce the same value as evaluating the entire subtree under the original triangle. 172

6.7 A class diagram showing the hierarchy of first-pass processors. 175

6.8 A class diagram showing the hierarchy of simple completers. 176

6.9 A class diagram showing the hierarchy of complex completers. 177

CHAPTER 1

INTRODUCTION

The need to address IR query processing efficiency arises from two distinct but compounding issues: the increase in available information and the development of more sophisticated models for queries. We first discuss the effects of increases in data size, and outline solutions often employed to deal with this problem. We then turn our attention to retrieval model complexity, and show the problems that arise as the retrieval model grows in size and/or complexity. We briefly examine the solutions used to date for this problem, and show that each of the solutions considered so far has limited application. We then describe how the aim of this thesis is to not only improve coverage of queries that we can dynamically optimize, but to also explore how to determine when these solutions can be brought to bear.

We then introduce the four contributions made by this thesis. The first three contributions are novel dynamic optimizations. Each optimization is designed to handle a unique difficulty encountered when processing queries with complex structures. The final contribution is a fresh approach to query processing, based on adaptively applying dynamic optimizations based on the characteristics exhibited by the query at hand.

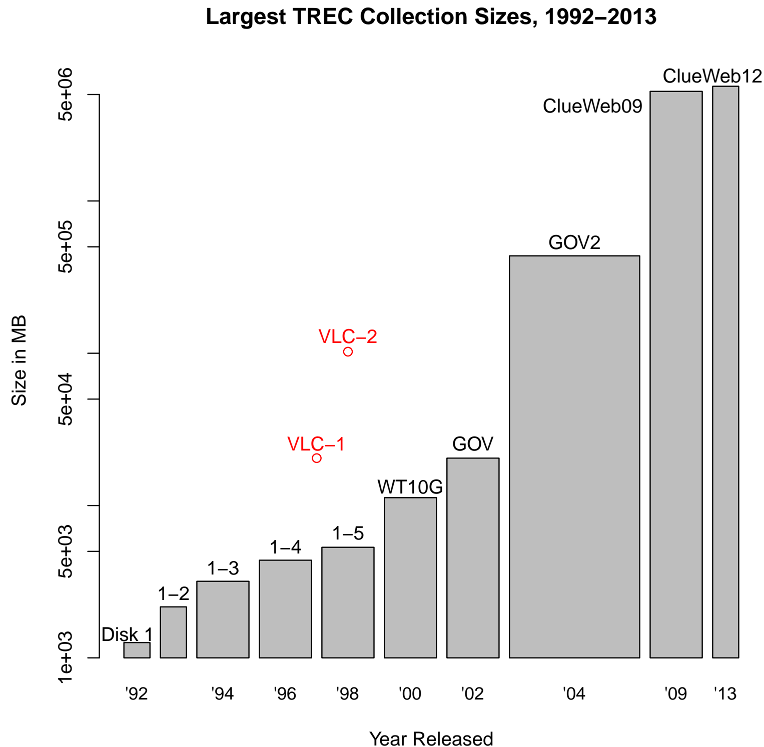

Figure 1.1. Growth of the largest single collection for a TREC track, by year. The width of the bar indicates how long that collection served as the largest widely used collection.

1.1 Problem: Bigger and Bigger Collections

In 1960, when information retrieval began to coalesce into a science in its own right, some early research in the field used collections as small as 100 documents over a single major topic (Swanson, 1960). Within a few years, researchers pushed to collections breaking the 1,000 document barrier, such as work conducted by (Dennis, 1964), and the Cranfield collection of 1,400 documents, as reported by Harman (Harman, 1993). Collection sizes have steadily increased since that time. An il-

lustration of this trend can be seen in the creation of the publicly reusable TREC collections, as shown in Figure 1.1. The data points indicate the largest popular collections released by TREC at the time. However as early as 1997 and 1998, the Very Large Collection (VLC) tracks investigated research using collections considerably larger than the typical research collection available at the time. The VLC collections (VLC-1 and VLC-2) saw less use in research outside of TREC, and therefore are not considered as part of the trend directly; their data points are shown for comparison. Even without considering the VLC collections, the super-linear increase in collection size over the years is clear, with the most recent data point occurring in 2013 with the release of the ClueWeb12 data set[1].

The ClueWeb12 collection represents an interesting shift in the targeted research of the IR community. Most, if not all, of the previous collections were made under a concerted effort to increase the scale and fidelity of the collection over the previous incarnations. ClueWeb12, in terms of pure web document count, is slightly smaller than ClueWeb09. However ClueWeb12 includes all tweets mentioned in the main web collection, as well as full dumps of the English subsets of Wikipedia and Wiki-Travel, and an RDF dump of Freebase, as separate auxiliary data sources meant to add structure, particularly named entities, to the main collection. These additional data sources make the ClueWeb12 collection "bigger" than ClueWeb09 by increasing the dimensionality of the collection; the extra data sources allow for a significantly denser set of relationships between the documents. Researchers can now investigate

[1]http://lemurproject.org/clueweb12.php/

implicit relationships of entities between the auxiliary data sources and the main web collection, in addition to the explicit hyperlink references in the web documents alone. In addition, such a collection suggests the notion of retrieval over entities, such as people or locations.

Outside of the research community, the growth in both size and complexity has been substantially more rapid. As of March 7, 2012, a conservative estimate of the size of Google's index is over 40 billion web pages (de Kunder, 2012), meaning even the largest research collection available today is less than 3% of the scale dealt with by the search industry. In terms of complexity, industry entities must contend daily with issues such as document versioning, real-time search, and other in-situ complications that are still outside of the scope of most non-industry researchers today.

As a simple example, Table 1.1 shows the runtime for both simple disjunctive keyword queries (KEY) and queries with a conjunctive component (SDM). Even at what is now a modest collection size of 3 million documents, SDM processing times begin to reach times that are unacceptable for commercial retrieval systems (Schurman & Brutlag, 2009). According to this anecdotal evidence, without further treatment the only hope to maintain efficient response times is to sacrifice the increase in effectiveness afforded by the more complex SDM.

As we can see, substantial evidence from academia and industry suggest that "big data" in IR is not only here to stay, but that the trend towards increasingly larger collections will only continue. Therefore scientists must develop solutions to manage

# Docs (M)	DIS	SDM
1	65	174
2	123	330
3	175	472
4	221	598
5	265	716
6	310	836
7	355	958
8	403	1083
9	449	1210
10	490	1325

Table 1.1. Execution time per query as the active size of a collection grows, from 1 million to 10 million documents. The first 10 million documents and first 100 queries from the TREC 2006 Web Track, Efficiency Task were used. Times are in milliseconds.

data of this magnitude in order for research to realistically progress. We review some of these solutions now.

1.1.1 Solutions

Historically, researchers have dealt with increasing collection sizes by developing techniques that avoid dealing with the whole collection. A simple example of such a technique is *index-time stopping*: certain words designated as stopwords are simply not indexed, so if the word occurs in the query it can be safely ignored and has no influence on the ranking of the document. Stopping has the added benefit of significantly reducing index size, as stopwords are typically the most frequent terms that occur in the given collection.

Recent advances in optimization techniques to address new complexity issues have mostly consisted of offline computation, such as storing n-grams for use during query

time. However, static optimization solutions do not fully address the problems; often many queries are left unimproved due to the space-coverage tradeoffs that must be made. Alternatively, the state-of-the-art techniques in dynamic optimization have only recently begun to receive attention, but even these new methods for the time being are ad-hoc and only target issues that arise in specific retrieval models.

In the case where the index is a reasonable size (i.e. can be kept on a single commodity machine), solutions such as improvements in compression technology (Zukowski, Heman, Nes, & Boncz, 2006), storing impacts over frequency information (Anh & Moffat, 2006; Strohman & Croft, 2007), document reorganization (Yan, Ding, & Suel, 2009a; Tonellotto, Macdonald, & Ounis, 2011), pruning algorithms, both at index- and retrieval-time (Turtle & Flood, 1995; Broder, Carmel, Herscovici, Soffer, & Zien, 2003; Büttcher & Clarke, 2006), and even new index structures (Culpepper, Petri, & Scholer, 2012), have all provided substantial gains in retrieval efficiency, usually without much adverse cost to retrieval effectiveness.

However since the advent of "Web-scale" data sets, storing an index of the desired collection on a single machine is not always a feasible option. Advances in distributed filesystems and processing (Ghemawat, Gobioff, & Leung, 2003; Chang et al., 2008; Isard, Budiu, Yu, Birrell, & Fetterly, 2007; DeCandia et al., 2007) over the last 10 years or so have made it clear that in order to handle collections of web scale, a sizable investment in computer hardware must be made as well. In short, the most common solution to handling large-scale data is to split the index into pieces known as *shards*, and place a shard on each available processing node. The parallelism provided by this approach typically yields substantial speedups over using a

single machine. Since this solution was popularized, whole fields of research have dedicated themselves to examining the cost/benefit tradeoff of balancing speed and coverage against real hardware cost. In IR, this subfield is commonly called *distributed IR*; much of the research distributed IR has focused on how to best process queries on a system that involves a cluster of machines instead of a single machine. Popular solutions typically involve splitting up the duties of query routing, rewriting, document matching, and scoring (Baeza-Yates, Castillo, Junqueira, Plachouras, & Silvestri, 2007; Moffat, Webber, Zobel, & Baeza-Yates, 2007a; Jonassen, 2012; Gil-Costa, Lobos, Inostrosa-Psijas, & Marin, 2012).

As expected, as the size of the collection increases, so does the runtime. If we consider an index of 10 million documents (bottom row), and we wanted to shard across, say, 5 machines, the effective execution time reduces down closer to the times reported for 2 million documents - a savings of approximately 75% for both models. A modest investment in additional hardware can substantially reduce processing load per query, making this an attractive solution for those needing to quickly reduce response time for large collections.

A full exploration of this aspect of information retrieval is outside the scope of this thesis, so we assume that a collection is a set of data that can be indexed and held on a single computer with reasonable resources (i.e. disk space, RAM, and processing capability attainable by a single individual or small-scale installation). All of the solutions presented in this thesis should, with little to no modification, translate to the distributed index setting, where the contributions described here would be applied to single shard in a distributed index.

1.2 Problem: Bigger and Bigger Queries

Research in information retrieval models often involves enriching an input query with additional annotations and intent before actually scoring documents against the query. The goal is to have the extra information provide better direction to the scoring and matching subsystems to generate more accurate results than if only the raw query were used. These improvements often require additional execution time to process the extra information. Recent models that have gained some traction in the last decade of IR research involve n-gram structures (Metzler & Croft, 2005; Bendersky, Metzler, & Croft, 2011; Xue, Huston, & Croft, 2010; Cao, Nie, Gao, & Robertson, 2008; Svore, Kanani, & Khan, 2010), graph structures (Page, Brin, Motwani, & Winograd, 1999; Craswell & Szummer, 2007), temporal information (He, Zeng, & Suel, 2010; Teevan, Ramage, & Morris, 2011; Allan, 2002), geolocation (Yi, Raghavan, & Leggetter, 2009; Lu, Peng, Wei, & Dumoulin, 2010), and use of structured document information (Kim, Xue, & Croft, 2009; Park, Croft, & Smith, 2011; Maisonnasse, Gaussier, & Chevallet, 2007; Macdonald, Plachouras, He, & Ounis, 2004; Zaragoza, Craswell, Taylor, Saria, & Robertson, 2004). In all of these cases, the enriched query requires more processing than a simple keyword query containing the same query terms.

The shift towards longer and more sophisticated queries is not limited to the academic community. Approximately 10 years ago, researchers found that the vast majority of web queries were under 3 words in length (Spink, Wolfram, Jansen, & Saracevic, 2001). Research conducted in 2007 suggests that queries are getting longer (Kamvar & Baluja, 2007), showing a slow but steady increase over the study's two-

year period. Such interfaces as Apple's Siri[2] and Google Voice Search[3] allow users to speak their queries instead of type them. Using speech as the modality for queries inherently encourages expressing queries in natural language. Additionally, growing disciplines such as legal search, patent retrieval, and computational humanities can benefit from richer query interfaces to facilitate effective domain-specific searches.

In some cases, the explicit queries themselves have grown to unforeseen proportions. Contributors to the Apache Lucene project have reported that in some cases, clients of the system hand-generate queries that consume up to four kilobytes of text (Ingersoll, 2012), although this is unusual for queries written by hand. Queries generated by algorithms (known as "machine-generated queries") have been used in tasks such as pseudo-relevance feedback, natural-language processing (NLP), and "search-as-a-service" applications. These queries can often produce queries orders of magnitude larger than most human-generated queries. Commonly commercial systems will often ignore most of the query in this case, however a system that naively attempts to process the query will be prone to either thrash over the input or fail altogether.

In both academia and industry, current trends indicate that the frequency of longer and deeper (i.e. containing more internal structure) queries will only continue to grow. To compound the problem, retrieval models themselves are also growing in complexity. The result is more complex models operating on larger queries, which can

[2]http://www.apple.com/iphone/features/siri.html

[3]http://www.google.com/mobile/voice-search/

create large processing loads on search systems. We now review several representative attempts at mitigating this problem.

1.2.1 Solutions

Optimization has typically progressed via two approaches. The first is *static* optimization, where efficiency improvements are made during index time, independent of a query that may be affected by such changes. The second is *dynamic* optimization, which occurs when the query is processed. This second group of techniques usually depends on the current query to influence decisions made during evaluation.

Referring to Table 1.1 again, we see that our more complex retrieval model (SDM) also benefits from sharding, however the execution time remains approximately three times slower than for the simpler KEY model. This suggests that in order to reduce the execution time to that of the KEY model, three times as many shards are needed to distribute the processing load. While the increase is not astronomical, this new cost-benefit relationship is nowhere near as attractive as the original ratio; while sharding can indeed help, the impact of increased complexity is still very apparent.

As these complex retrieval models are relatively new, most efficiency solutions that are not sharding-based so far are often ad hoc. A typical solution is to simply pretend the collection is vastly smaller than it really is, meaning a single query evaluation truly only considers a small percentage of the full collection. As an example, a model such as Weighted SDM (WSDM) (Bendersky et al., 2011) requires some amount of parameter tuning. Due to the computational demands of the model, it is infeasible to perform even directed parameterization methods like coordinate ascent, which may require hundreds or thousands of evaluations of each training query. In-

stead, for each query they execute a run with randomized parameters, and record the top 10,000 results. These documents are the only ones scored for all subsequent runs, and the parameters are optimized with respect to this subset of the collection (Bendersky, 2012). While this solution makes parameterization tractable, it is difficult to determine how much better the model could be if a full parameterization were possible.

Recent research in optimization has begun to address these new complexities, however the success and deployment of these solutions has been limited. As an example, consider the use of n-grams in a model such as the Sequential Dependence Model (SDM) by (Metzler & Croft, 2005). The SDM uses ordered (phrase) and unordered (window) bigrams; calculating these bigram frequencies online is time-consuming, particularly if the two words are typically stopwords (e.g., "The Who").

A common solution to this problem is to pre-compute the bigram statistics offline and store the posting lists to directly provide statistics for the bigrams in the query. However this approach must strike a balance between coverage and space consumption. A straightforward solution is to create another index of comparable size to store frequencies for the phrases. Often times these additional indexes can be much larger than the original index, so to save on space, the frequencies are thresholded and filtered (Huston, Moffat, & Croft, 2011). The end result is that as collections grow in size, a diminishing fraction of the bigram frequencies are stored. However to service all queries, the remaining bigrams must still be computed online. Storing n-grams of different size (e.g., 3-grams) exacerbates the problem, but may still be tractable via the heuristics mentioned earlier. Worse yet is the attempt to store the

spans of text where the words in question may appear in any order (commonly referred to as *unordered windows*), which are also used in the SDM. No known research has successfully pre-computed the “head” windows to store for a given window size, and the problem quickly becomes unmanageable as the value of n increases. In this case, the only feasible option is to compute these statistics at query time. In short, offline computation of anything greater than unigrams can only go so far, as the space of possible index keys is far larger than available computing time and storage.

Another possible solution can be to hard-wire computation as much as possible. In certain settings where an implementor has specialized hardware to compute their chosen retrieval model, the computing cost can be drastically reduced by pushing computation down to the hardware level. However, this approach requires pinning the system to a specific, and now unmodifiable, retrieval model. Such an approach also requires substantial resources along other dimensions (i.e. capital, access to circuit designers, etc), which many installations do not have.

Other popular solutions to this problem involve 1) novel index structures (Culpepper et al., 2012) and 2) treating computation cost as part of a machine learning utility function. Both approaches have shown promise, however both also have severe limitations to their applicability. The new index structures often require the entire index to sit in RAM, and despite advances in memory technology, this requirement breaks our commodity computer assumption for all but trivially-sized collections. The machine learning approaches inherit both the advantages and disadvantages of machine learning algorithms; they can tune to produce highly efficient algorithms while minimizing the negative impact on retrieval quality, however appropriate training data

must be provided, overfitting must be accounted for, and new features or new trends in data will require periodic retraining in order to maintain accuracy. In this thesis we focus on improvements to algorithmic optimizations. Therefore improvements in index capability readily stack with the improvements presented here, and no training is necessary to ensure proper operation.

1.3 Another Look at the Same Problem

We now see the two major dimensions of the efficiency problem: 1) collection sizes are growing, and 2) retrieval models are getting more complicated. An effective and scalable solution (to a point) for larger data sizes is to shard the collection over several processing nodes and exploit data parallelism. Several commercial entities have shown the appeal of using commodity hardware to provide large-scale parallelism for a reasonable cost, relative to the amount of data. While not a panacea to the data size problem, the approach is now ubiquitous enough that we will assume either we are handling a monolithic (i.e. fits on one machine) collection, or a shard of a larger collection. Therefore operations need only take place on the local disk of the machine.

In dealing with more complex retrieval models, no one solution so far seems to be able to address this problem. Indeed, the nature of the problem may not lend itself to a single strategy that can cover all possible query structures. Pre-computation approaches and caching provide a tangible benefit to a subset of the new complexity, but such approaches cannot hope to cover the expansive implicit spaces represented by some constructs, which means in terms of coverage, much of the problem remains.

Algorithmic solutions so far have limited scope; in some cases, the assumptions needed render the solution useless outside of a specific setting. Instead of focusing on one optimization in isolation, it may be time to consider query execution as something that requires planning to choose which optimizations should be applied to a particular query.

This thesis describes optimizations as behaviors that are exhibited by the various operators that compose a query in the retrieval system. An example behavior may be whether a particular operator's data source (where it gets its input) resides completely in memory, or is being streamed from disk. In the case of the latter, the system may decide to hold off generating scores from that operator if the cost/benefit of that operator is not high enough. Conversely, if the operator is entirely in memory, the system may always generate a score from that operator, as its disk access cost is zero. Using this approach, we can both easily 1) add new operators that exhibit existing behaviors to immediately take advantage of implemented optimizations, and 2) add new behaviors to existing operators to leverage advances in research and engineering.

1.4 Contributions

This thesis introduces three new dynamic optimization techniques based on leveraging query structure in order to reduce computational cost. Additionally, this thesis introduces a novel design approach to dynamic optimization of retrieval models, based on the attributes of the query components constructed by the index subsystem. The contributions of this thesis are as follows:

I We empirically show that queries can be automatically restructured to be more amenable to classic and more recent dynamic optimization strategies, such as the Cascade Ranking Model, or Selective WAND Pruning. We perform an analysis of two classes of popular query-time optimizations – algorithmic and machine-learning oriented – showing that introducing greater depth into the query structure reduces pruning effectiveness. In certain query structures, which we call "interpolated subqueries", we can reduce the depth of the query to expose more of it to direct pruning, in many cases reducing execution time by over 80% for the Maxscore scoring regime, and over 70% for the Weak-AND, or Wand, regime. Finally, we show that the expected gains from query flattening have a high correlation to the proportion of the query that can be exposed by the flattening process.

II We define a new technique for alternative formulations of retrieval models, and show how they provide greater opportunity for run-time pruning by following a simple mathematical blueprint to convert a complex retrieval model into one more suitable for the run-time algorithms described in contribution I. We apply this reformulation technique to two popular field-based retrieval models (PRMS and BM25F), and demonstrate average improvements to PRMS of over 30% using the reformulated models.

III We introduce the "delayed execution" optimization. This behavior allows for certain types of query components to have their score calculations delayed based on their complexity. We demonstrate this optimization on two basic term conjunction scoring functions, the previously mentioned ordered window and un-

ordered window operations. The delayed execution of these components allows us to complete an estimated ranking in approximately half the time of the full evaluation. We use the extra time to explore the tradeoff between accuracy and efficiency by using different completion strategies for evaluation. We also exploit dependencies between immediate and delayed to reduce execution time even further. In experiments using the Sequential Dependence Model, we see improvements of over 20% using approximate scoring completion techniques, and for queries of length 7 or more, we see similar improvements without sacrificing score accuracy. We also test this method against a set of machine-generated queries, and we are able to considerably improve efficiency over standard processing techniques in this setting as well.

IV We introduce Julien, a new framework for designing, implementing, and planning with retrieval-time optimizations. Optimization application is based on exhibited behaviors (implemented as traits, or mixins) in the query structure, instead of relying on hard-coded logic. We show that the design of Julien allows for easy extension in both the directions of adding new operators to the new, and adding new behaviors for operators that the query execution subsystem can act upon. As further evidence of the effectiveness of this approach, we implement the previous contributions as extensions to the base Julien system.

1.5 Outline

The remainder of this thesis proceeds as follows. In Chapter 2, we review the evolution of optimization in information retrieval. We then conclude with a review

of four popular dynamic optimization algorithms for ranked retrieval. Chapter 3 presents the query depth analysis of the four algorithms, and we empirically show the benefits of query flattening. In Chapter 4 we introduce the alternative scoring formulation, and demonstrate its effectiveness on two well-known field-based retrieval models. Chapter 5 then presents delayed evaluation, which enables operators to provide cheap estimates of their scores in lieu of an expensive calculation of their actual scores. After initial estimation, we investigate several ways to complete scoring while using as little of the remaining time as possible. In Chapter 6, we present Julien, a retrieval framework designed around the behavior processing model. We implement the three optimizations in Julien, allowing the improvements to operationally coexist in one system; an important step often overlooked in other optimizations. We then verify the previous discoveries by recreating a select set of experiments, and show that the trends established in previous chapters hold when applied in a peer system. The thesis concludes with Chapter 7, where we review the contributions made, and discuss future extensions using the advances described in this thesis.

CHAPTER 2

BACKGROUND

This chapter serves both to inform the reader of general background in optimization in Information Retrieval, and to introduce the assumptions and terminology used in the remaining chapters of the thesis. We first introduce the terminology in use throughout this work. We proceed with a review of relevant prior work in optimization, culminating in a description and assessment of two classes of state-of-the-art dynamic pruning techniques used across various retrieval systems: algorithmic approaches, represented by the Maxscore and Weak-AND (WAND) algorithms, and machine learning approaches, represented by the Cascade Rank Model (CRM), and the Selective Pruning Strategy (SPS).

We then review several popular web and research retrieval systems to determine the current operations supported by these systems. This assessment lays the groundwork for approaching query processing from a behavioral standpoint, which we address in depth in Chapter 6.

2.1 Terminology

Many of the techniques described here generalize over retrieval types and index-specific implementations. Towards this end, we introduce a generic vocabulary to avoid the implicit assumption that these techniques only apply to text documents.

Let a *retrievable object* (or *retrievable*) be any object that we will want to see in a response to a query. Examples of retrievables are a text document, a recognized entity such as a person, or a video. Therefore, ranked lists consist of a ranking of retrievables, ordered from most to least likely relevant, with respect to a given query. An *index key* (or *key*) is a piece of information we can extract from a retrievable such that the key may be used in a query. Keys are typically tokens either directly observed or extracted from a retrievable, although other keys, such as pixel color and intensity, could be used.

We define an *index* as a mapping from the space of index keys to retrievables. More specifically, every index key maps to a (possibly sorted) set of tuples, which we will call *postings*, each of which containing a reference to a retrievable and additional information describing the relationship between that key and the retrievable.

A *collection* is a set of retrievables. While this set may be infinite (e.g. a real-time Twitter stream), in the context of this thesis we limit the discussion to finite collections. The *vocabulary* is the set of index keys extracted during the indexing process. Note that while a different set of keys may be extracted during any given indexing run, the final set of keys is finite, since the collection the keys are extracted from is finite (we do not allow infinite generation of index keys from a single retrievable source).

Finally, we define a *retrieval universe* as a mathematical structure containing one or more collections of retrievables, their extracted keys, and a set of associated operators which can be used to map between index keys and retrievables.

In the classic information retrieval setting, the retrieval universe consists of a single collection, where the retrievables are text documents. Each document contains sequences of words as its content, and each word, in a possibly normalized form, is extracted from the document and used as an index key. Therefore the vocabulary is the set of all unique terms from all documents in the collection. In this setting, the index maps a word to a list of postings, where each posting is a reference to the retrievable (document identifier), how many times the key occurs in the retrievable (term count), and possibly the locations of the occurrences in the retrievable (document positions).

For the remainder of the thesis, Q will refer to the user-supplied query, d will refer to a retrievable to be scored from a collection of D retrievables, and R will refer to the ranked list of retrievables after a query has been processed.

2.2 A Brief History of IR Optimization

We begin this section by displaying a reference for the following discussion on optimization in IR. In Figure 2.1 we present the standard set of steps taken by a modern retrieval system in order to process an information need. The information need begins as an abstract notion of some information the user (or system) does not have, but would like to. In Figure 2.1, the example information need we have is "What has the Hubble telescope discovered?" From this point, the user carries out

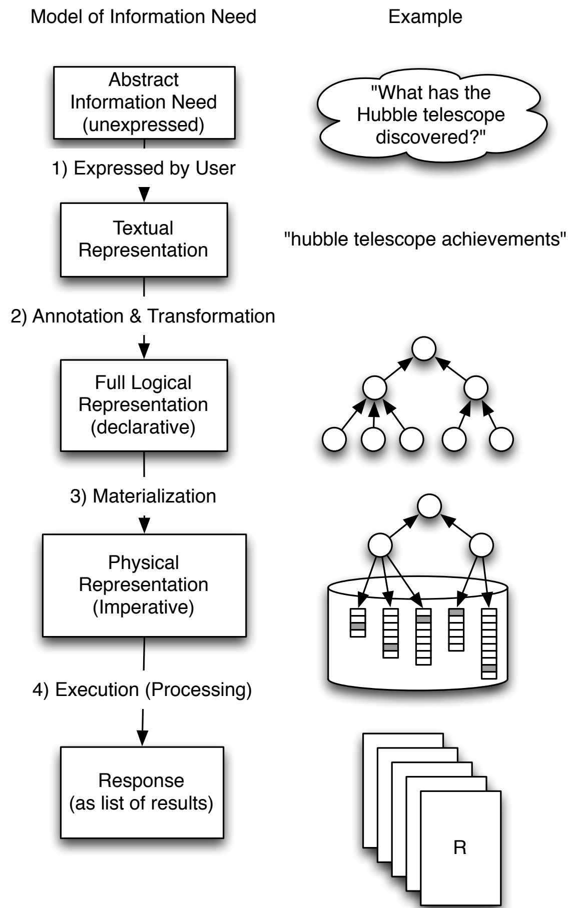

Figure 2.1. The standard steps taken to process a query, from input of the raw query to returning scored results.

step (1): he or she produces a representation of that information need in the form of a query - some expression of the information need to the retrieval system that the user thinks will generate a response that will fulfill the information need. In our example, the user issues the query "hubble telescope achievements" which they believe will produce an appropriate response. Queries are typically expressed to the system via a query language, which allows the user to impose a certain amount of logic to their query to better model their information need. The next step (2), usually taken by the system, is to annotate, transform, and otherwise enrich the query into some internal logical representation an in attempt to make the query a more explicit expression of the information need, and to prepare the query for materialization, which is the next step. Materialization involves turning the logical representation of the query into a set of objects that the system can use to score documents. This physical manifestation of the query can then be processed over the index to produce a final ranked list of results, which is (hopefully) the desired response from the system that fulfills the user's original information need. Due to the human interaction required in step 1, optimizations to this process can only take place in steps 2-4.

The implementation of the three optimizations discussed in this thesis can easily be placed into the process shown in Figure 2.1. Flattening query structure (Chapter 3) can be cleanly implemented in step 2, during transformation of the textual query into the logical representation. Alternative Scoring Representations (Chapter 4) require implementation in both steps 2 and 3; in step 2 we recognize a replaceable query (or subquery), and during transformation, replace that part of the query with the appropriate alternative expression. In step 3, we must make the new

scoring functions available in order to materialize them. Oftentimes implementing these scoring functions is done offline, but the effect of this change is seen during this step. Finally, Delayed Execution (Chapter 5) requires changes to steps 2, 3, and 4. The system must recognize and replace the appropriate structures in the logical representation (step 2), have access to the correct scoring functions for materialization (step 3), and the processing subsystem must be able to handle regular and estimated scoring calls, and must proceed in two stages (step 4). We discuss these implementations in detail in Chapter 6.

We now introduce the commonly used terminology and the subset of optimization developments most relevant to the research described in this thesis.

2.2.1 Processing Models & Index Organizations

The first distinction we make involves the processing model (also known as the "scoring regime"). The scoring model determines to some degree what dynamic optimizations are possible. The predominant models are: Document-At-A-Time (DAAT), Term-At-A-Time (TAAT), and Score-At-A-Time (SAAT). DAAT instantiates all index pointers required by the query at one time, and synchronizes data reads off the index pointers in order to completely score a document at one time. No more than k documents are retained in memory at any time, however all index pointers must be active at the same time. The DAAT model requires that the posting lists in the index be sorted in document-id order, otherwise the system spends an inordinate amount of time randomly seeking for document ids within the posting lists.

TAAT scoring opens one index pointer at a time, scoring all documents found in the posting list and saving results in an accumulator, then moving on to the next

index pointer. TAAT may be implemented over a document-id sorted index, or a score-sorted index. The number of pointers open at one time is minimal (1), however an accumulator structure is required to completely score documents.

Finally, SAAT relies on score-sorted (also referred to as impact-sorted) posting lists in order to properly function. The system opens all index pointers simultaneously, and reads from the head of the posting list that has the score with the largest impact first. The pointers are kept in a heap structure, and are read until all lists are exhausted. SAAT scoring also requires an accumulator structure, and that all index pointers be opened at the beginning of scoring.

Although a significant amount of work has covered posting list optimization for impact-sorted processing strategies, our focus lies on regimes that operate on a document-sorted index, as other organizations provide poor support for query operations that involve dependencies between index keys (e.g. phrase or window operations). Additionally, they have been shown to have problems when scaling up to large collections due to the accumulator footprint growing linearly with the size of the collection.

We consider single token nodes as atomic nodes (i.e. "terms"). Therefore in TAAT or SAAT scoring, when a unigram appears independently and also as part of an n-gram node, we would have to either traverse its posting list multiple times, or cache the positions for the list until we read the other terms involved in the n-gram. Since large, structured queries are the target of this thesis, this problem will most likely occur in every query encountered. In general, any method that must use accumulators encounters the issue of the memory footprint becoming an additional

limiting factor, and we have found no research that shows that TAAT or SAAT organizations have superior retrieval efficiency over DAAT with queries that involve multi-term dependencies.

2.2.2 Optimization Attributes

We now describe several classification axes that we can use when describing optimization techniques in Information Retrieval.

Index-time vs. Query-time. The first axis of classification is whether an optimization can be applied without rebuilding the index. A *static* (or *index-time*) optimization must be applied at index time. Therefore if the optimization was not previously applied, or its parameters change, the index must be rebuilt in order for the optimization to be effective. A simple example of such an optimization is using stopwords – at index time we do not index stopwords in order to reduce index size and speed up query evaluation at retrieval time. However if we change the list of stopwords, we must re-index to reflect that change. Conversely, a *dynamic* (or *query-time*) optimization can be applied at retrieval time, without the need to rebuild the index. An example of dynamic optimization is using a pruning algorithm such as Maxscore to stop scoring documents early, as the algorithm determines that no more documents will enter the result set. Another way to view this axis is *query-independent* vs. *query-dependent*. An optimization made at index time will be query-independent, as no query exists to make optimization decisions against. Conversely, a query-dependent optimization must be done at query-time, when a query is available.

Scoring safety. An optimization technique, when applied to query evaluation, may result in changes to the scores produced for a given document, or may omit

certain documents from the final result list. If an optimization does not modify the score produced when evaluating Q against any d in the collection, that optimization is *score-safe*. While an algorithm may produce the correct score for a given $\{Q, d\}$ pair, the algorithm may also opt to simply not score that pair, which produces a different final ranked list. If an optimization does not modify the order of documents after evaluating query Q, then the optimization is *rank-safe*. A weaker guarantee is that an optimization is *safe-to-rank-k*, meaning safety is guaranteed to rank k, where k is usually set by the client. Note that it is possible to have an algorithm that is score-safe, but not rank-safe, and vice-versa. When defining safety to a particular k, we may also define an even milder form of safety: *set-safe-to-rank-k*. This property guarantees that while the scores and ranks of the top k documents may be different from the original algorithm, the top k set is the same top k as the original algorithm.

Dependence-Aware. Many earlier optimizations focus on improving performance on traversing a single posting list, or they focus on accessing those lists as little as possible. In both cases, there is an implicit assumption that index keys in the query are all independent of each other. More recent research has attempted to address the independence assumption, therefore we differentiate between optimizations that are dependence-aware, and those that are not.

Parameterized. More recent optimization research has produced techniques that have parameters that require tuning. We differentiate between optimizations that make use of an algorithm to automatically parameterize (*parametric optimizations*) and those that do not (*non-parametric optimizations*).

2.2.3 Optimizations in Information Retrieval

Buckley and Lewit (Buckley & Lewit, 1985) described a set-safe dynamic optimization for TAAT evaluation in the SMART retrieval system. This is one of the earlier works of optimization specifically for an information retrieval system. The authors first note the idea of non-safe optimizations here, recognizing a pattern that holds true in many situations today: optimizations can often be dramatically more effective if they do not need to ensure score-safety.

Turtle and Flood first describe the Maxscore algorithm after a review of DAAT and TAAT scoring regimes (Turtle & Flood, 1995). The authors show that Maxscore can operate over both regimes, provided the index model supports skips over postings within a given term posting list and the system can provide estimates of the upper bound score that may be produced by a posting list. We call a function which provides such an estimate an *upper-bound estimator* (UBE). We will discuss the state of UBEs in Section 2.3.1.1.

Soon after the original Maxscore description, Turtle et al. created and patented a variation of the algorithm that makes use of an iterated sampling technique to handle complex query components that lack aggregate statistics in the index (Turtle, Morton, & Larntz, 1996). The patent itself provides a detailed description of the sampling process, but due to the nature of the publication, no experimental results are present, so it is difficult to judge the technique directly against other methods. Although the methods presented in Chapter 5 were developed independently, and before this approach was known, it is worth noting that this approach is philosophically similar, in that both techniques attempt to reduce the work involved in computing

complex query components (specifically phrase and window operators in this case). Whereas the work of Turtle et al. performs pre-sampling or iterated sampling, the techniques we present make no attempt to gather aggregate statistics until after a prelminary pass over the collection is complete. We hypothesize that post-processing the statistics should provide better opportunity for eliding work (since no preliminary work is done), but leave this exploration for future work.

(Strohman, Turtle, & Croft, 2005) combined the Maxscore technique of Turtle and Flood and the top-documents caching mechanism of (Brown, 1995) to further decrease the evaluation time of queries. The authors claim that the newer optimization returns the exact list of results as an unoptimized evaluation while gaining a 61% increase in efficiency over an unoptimized DAAT approach. A limitation of the approach shown, however, is that the method locks the index into a particular retrieval model, whereas Maxscore alone can be applied to virtually any model at retrieval time without reindexing. While Maxscore is a pure dynamic optimization, the Topdocs extension adds a static component that limits the flexibility of the optimization.

Broder et al. (Broder et al., 2003) described a two-pass algorithm over DAAT indexes they dubbed WAND, which reduces the number of postings to decode by first considering any given query to be a strictly conjunctive query, then relaxing the constraint as necessary to complete scoring. Like Maxscore, WAND relies on upper-bound estimators to operate efficiently. The authors describe several UBEs for unigrams and bigrams, however they admit their estimators could be improved upon. Similar to the work by (Buckley & Lewit, 1985), the authors here also note that

if the rank-safety constraints are relaxed, the WAND algorithm saves significantly more than when it operates in strict rank-safe mode.

(Büttcher & Clarke, 2006) investigate improvements afforded using static index pruning in order to create an index small enough to fit into main memory of a machine. Their technique was document-centered, versus a previous approach by (Carmel et al., 2001) which focused on term-centered static pruning. Both techniques use KL-divergence combined with some form of threshold in order to determine what elements of the index should be omitted, concluding that a document-centric pruning approach, dropping some amount of each document, was the most effective at balancing the efficiency/effectiveness tradeoff. The approach produced a very small and fast index that could be used in main memory, with a full index available on-disk for backoff. Note that none of these techniques have been shown to be rank-safe, as they must make pruning decisions independent of queries. However these techniques are applicable across different scoring models, as they involve decisions of which key/retrievable mappings to store in the index, and not the content of the stored postings themselves.

Another avenue of static optimization involves reorganizing the input data to the indexing system in order to improve compressed index size or query throughput (Blandford & Blelloch, 2002; Silvestri, 2007). The intuition behind this line of research lies in how most retrieval systems construct indexes—most indexers assign internal numbers to the documents as they enter the system. The compression efficiency, and consequently the decompression and reading efficiency, of the resulting index may heavily depend on the order of the documents input to the indexer.

Supplemental to this is an investigation by Yan et al. (Yan, Ding, & Suel, 2009b) of several compression strategies for maximizing query throughput given a re-order input sequence to the indexer. Their results suggest that different encoding schemes could be used on different sections of the index to find an optimal tradeoff between index size and speed.

Anh et al. introduced the notion of quantized weights, or binning, the contributions made by the terms in documents (Anh, de Kretser, & Moffat, 2001). These modifications provided significant gains in retrieval efficiency for independent keyword models. They later developed several new dynamic optimization techniques based on impact-sorted indexes (Anh & Moffat, 2006). They also provide an excellent overview of index models and optimization strategies to date.

Strohman and Croft extend the dynamic optimization of bins even further by not only terminating the phase where new accumulators can be added (the OR-mode), but during the AND-mode, iteratively reducing the number of active remaining bins (Strohman & Croft, 2007). This resulted in having fewer bins to update and a reduced sort time at the end.

Note that all of the discussions so far assume a simple keyword query structure. Therefore components of the query (typically individual index keys) are assumed to behave independently, and their co-occurrence has no added effect on scoring. We now turn to more recent optimizations that are dependence-aware.

Researchers began this exploration by investigating ways to precompute and store n-gram information at index time, which mitigates the cost of performing posting list intersection at retrieval time (Zhu, Shi, Li, & Wen, 2007; Schenkel, Broschart,

Hwang, Theobald, & Weikum, 2007; Yan, Shi, Zhang, Suel, & Wen, 2010; Huston et al., 2011). These optimizations are all rank- and score-safe; the tradeoff made here is the use of extra disk space and index time to save on computation of n-gram occurrences at retrieval time. No method to date can feasibly store all combinations of requested n-grams, therefore improvements only occur on a selected subset of queries, although through query log analysis the queries can be selected to form a large percentage of an expected query stream.

Zhu et al. performed an interesting set of experiments that incorporated both term proximity and document structure (Zhu et al., 2007). Their conclusions only made minor changes to index structure, yet they benefited greatly from considering these more complex factors. Similar work was explored by (Schenkel et al., 2007).

More recently, Yan et al. created a supplemental term-pair index in order to increase efficiency in processing proximity-aware queries (Yan et al., 2010). The approach showed that if one considers term dependencies up to three terms away[1], and the correct term-pairs are chosen, that a considerable amount of processing time can be elided.

Researchers have also taken an interest in leveraging machine learning techniques to tune the optimization of a system to an even higher degree (Wang, Lin, & Metzler, 2011; Wang, Metzler, & Lin, 2010; Wang, Lin, & Metzler, 2010; Tonellotto, Macdonald, & Ounis, 2013). Two of the models in this class are the focus of further discussion, so we postpone describing them in detail to Section 2.3.2.

---

[1] If the term pair is $t_1 t_2$, then three terms away is any 5-gram such that $t_1 t_x t_y t_z t_2$.

As we can now see, there are numerous places in the retrieval pipeline, both offline and online, to affect the efficiency of the retrieval process. We now briefly survey modern day retrieval systems to establish what operations search must support, which in turn will define what behaviors we wish to express to the processing subsystem.

2.3 State-of-the-art Dynamic Optimization

We now provide an intuition for dynamic optimization, and follow with a review of two classes of dynamic optimization: algorithmic and machine-learning based techniques.

To provide an operational description of query execution, we use pseudo-code where needed. Consider the exhaustive scoring regime for DAAT processing - every candidate from every posting list accessed is scored. Algorithm 1 describes this process. Q is the supplied query, and I is a reference to the index that Q should be evaluated against. R is a priority queue sorted on increasing document score, meaning the "head" of the queue is the document in the queue with the lowest score. Scored documents are inserted into the queue as $\langle$ document id, score $\rangle$ tuples, as shown in line 9. At the end of scoring, we reverse the order of R (line 13) and return R', which contains the top k results in decreasing score order. Every document containing at least one query term is completely scored and considered for R. The only filtering performed is when a scored document d is not added to R if the lowest score in R is already higher than d's score (line 8). The size of R is kept limited to k

Algorithm 1 An exhaustive DAAT scoring algorithm.

```
ExhaustiveScoreDocuments(Q, I, k, R)
 1  R = {}
 2  θ = −∞
 3  while Q has unfinished terms
 4      d = MinimumCandidate(Q, I)
 5      s_d = 0
 6      foreach q ∈ Q
 7          s_d = s_d + Score(d, q, I)
 8      if s_d > θ
 9          Insert(R, ⟨docid = d, score = s_d⟩)
10          if |R| > k
11              Dequeue(R)
12              θ = R.head.score
13  R' = Reverse(R)
14  return R'
```

in lines 10-11, but this check comes after the work has already been done to update the queue.

From this simple description, it is clear that if k is small compared to the number of candidates, then this algorithm wastes an immense amount of work scoring documents that will not even appear in the result queue. Dynamic optimization aims to mitigate this issue as much as possible.

2.3.1 Algorithmic Dynamic Optimization

We now introduce and describe two highly effective and well-studied algorithms that operate under a DAAT processing model. We use the following to establish a solid understanding of these algorithms, which will motivate further discussion in

the thesis. In the following, we assume that k results have been requested, and the retrieval model is some form of simple (i.e., disjunctive) keyword query, meaning each query component is one-to-one mapped to an index key, and their score contributions are calculated independently. We also assume that both algorithms only operate on the scoring components directly under the root of the query graph; no nested pruning operations are considered. Also of note is that the query components are commutative: regardless of their order of evaluation, calculating and summing the query components results in the same score for a particular retrievable.

Before moving to the algorithms themselves, we introduce the notion of an *upper bound estimator*, an important construct in the following algorithms.

2.3.1.1 Upper Bound Estimators

Upper Bound Estimators (UBEs) are functions or stored values that provide an estimate of the maximum contribution a query component may make during query evaluation. Historically, there has been a one-to-one correspondence of one UBE for every index key in a query. However we may create UBEs for components such as ordered windows or field-dependent components. We call a UBE *admissible* when no actual value generated by the component is greater than the UBE. A useless yet admissible UBE for any scoring algorithm is $+\infty$.

Initial UBEs were offered by (Turtle & Flood, 1995) and (Broder et al., 2003), both of which simply used the stored maximum frequency of each term, transformed by the scoring function of interest to generate the UBE value. However in both cases they did not provide any formal justification for using this value. (Macdonald, Ounis, & Tonellotto, 2011) provided this justification, officially naming the UBE as

max_{tf}. They derive the max_{tf} estimator for Language Models, BM25, and DLH13, by viewing the estimator as a constrained maximization problem (CMP) over the space of term-frequencies and document lengths.

2.3.1.2 Maxscore

Algorithm 2 shows the pseudo-code operation of Maxscore. For a query Q, Maxscore makes use of $\text{UBND}(q, I)$, an admissible upper bound estimator (UBE) for each index key q. Maxscore also maintains a θ value, which is the lowest score in the list of scored candidates so far. If the number of documents scored so far is less than k, the desired number of ranked documents to return, then $\theta = -\infty$. The function INCDFSORT sorts the scorers in Q by increasing document frequency order. Therefore, the scorers with the least number of candidates are evaluated first.

The helper function SETSENTINELS utilizes θ to determine the minimal set of scorers that must have a hit on a given document to have a chance of producing a candidate score. SETSENTINELS simulates scoring a candidate document where each scorer misses the document. As the scorers are evaluated (i.e. generating a score of $\text{LBND}(Q[i], I)$ for scorer i), the score drops from the initial value of U_{Σ}. If the score drops below θ, then we know that that we cannot afford to have the first i scorers miss a document. Knowing this, we can drive candidate selection by only considering the first i scorers (since at least one must hit in order for the candidate to have a chance of entering R). Therefore, whenever $|SN| < |Q|$, Maxscore is skipping useless candidate documents that the set $\{Q - SN\}$ would otherwise generate in EXHAUSTIVESCOREDOCUMENTS. The goal of the sort on the scorers imposed by

Algorithm 2 Maxscore DAAT scoring algorithm.

MAXSCORE(Q, I, k)

1. $Q = \text{IncDFSort}(Q, I)$
2. $SN = Q$
3. $U_\Sigma = \sum_{q \in Q} \text{UBnd}(q, I)$
4. $R = \{\}$
5. $\theta = -\infty$
6. **while** SN has unfinished terms
7. $\quad d = \text{MinimumCandidate}(SN, I)$
8. $\quad s_d = U_\Sigma$
9. $\quad$ **foreach** $q \in SN$
10. $\quad\quad s_q = \text{Score}(d, q, I)$
11. $\quad\quad s_d = s_d + s_q - \text{UBnd}(q, I)$
12. $\quad$ **foreach** $q \in \{Q - SN\}$
13. $\quad\quad$ **if** $s_d < \theta$
14. $\quad\quad\quad$ abandon scoring of document d, and
15. $\quad\quad\quad\quad$ resume from step 6
16. $\quad\quad s_q = \text{Score}(d, q, I)$
17. $\quad\quad s_d = s_d + s_q - \text{UBnd}(q, I)$
18. $\quad$ **if** $s_d > \theta$
19. $\quad\quad \text{Insert}(R, \langle docid = d, score = s_d \rangle)$
20. $\quad\quad$ **if** $|R| > k$
21. $\quad\quad\quad \text{Dequeue}(R)$
22. $\quad\quad\quad \theta = R.head.score$
23. $\quad\quad\quad SN = \text{SetSentinels}(Q, U_\Sigma, \theta)$
24. $R' = \text{Reverse}(\text{R})$
25. **return** R'

SETSENTINELS(Q, U_Σ, θ)

1. $s = U_\Sigma$
2. $i = 1$
3. **while** $s > \theta$ and $i \leq |Q|$
4. $\quad s = s - \text{UBnd}(Q[i], I) + \text{LBnd}(Q[i], I)$
5. $\quad i = i + 1$
6. **return** $Q[1\,..\,(i-1)]$

IncDFSort is to maximize the number of skipped candidates from $\{Q - SN\}$. We reexamine this approach in Section 3.4.

At the beginning of scoring a real candidate d, the system begins from the maximum possible score U_Σ. The first inner loop, lines 9-11, directly calculates the partial score of the sentinels (line 7). As each term is evaluated for that document, its UBE is replaced by its actual score: $s_d = s_d + s_t - \text{UBnd}(t, I)$ (lines 10-11). Note that these successive replacements act as a nonincreasing estimate of the score for d. After the sentinels are complete, we move to the remaining scorers, but now check to see if the partial score has dipped below the threshold (lines 12-17). Therefore each replacement, if $s_d < \theta$, then we know d cannot make it into the final ranked list, and stop scoring d. Therefore Maxscore looks to short-circuit score calculation at the query component level.

2.3.1.3 Weak-AND

Weak-AND, or WAND for short, was proposed by (Broder et al., 2003) as a way to fully skip scoring documents using a two-level processing mechanism. Like Maxscore, WAND depends on admissible UBEs of each index key. When considering a candidate d, WAND first uses the UBEs to consider the query in a conjunctive manner, which determines whether or not d will be fully scored. Use of the PositionSort function, which sorts the scorers in increasing candidate order (i.e. the first scorer will have a candidate for scoring with the lowest internal docid).

The threshold θ is used to determine how conjunctive the query is considered to be. θ begins as $-\infty$, meaning any document qualifies for scoring. As documents are scored, θ increases, which in turn increases the need for candidates to have more and

more of the query present to qualify for full scoring. This requirement is enforced via the FindPivot helper function. As opposed to Maxscore, WAND looks to short-circuit scoring of whole candidates using the θ threshold. The quantity L_Σ is the lower bound sum of all the scoring components - therefore it is the lowest score the system can expect to receive from the scorers.

Additionally, the authors intended θ to be increased by an externally set multiplicative factor, α. Boosting the value of θ prematurely would cause fewer candidates to be fully scored earlier. Therefore this factor α provides a trade-off between speed and accuracy. At $\alpha = 1$, no accuracy is sacrificed, but speed up is limited. As α increases, fewer candidates are scored which can dramatically increase performance. However the lower α gets, the greater the probability that the algorithm produces a non rank-safe result. Note that if a document is scored, it is correct, so the algorithm is score-safe.

Algorithm 5 shows pseudo-code describing the operation of WAND. Although (Broder et al., 2003) originally describe the algorithm as an iterator that can be used over lower-level scoring iterators, the algorithm is described here as a scoring regime over iterators.

Both algorithms maintain an ordered list of posting list pointers, which we call the *sentinels*, used to cull the number of candidates. Based on θ, a minimum number of terms must be present in order for a candidate to be fully scored. For Maxscore, as θ increases (due to more candidates being scored), the sentinel list begins to shrink because fewer pointers must be checked to determine candidacy (imagine when missing only one term causes $s_d < \theta$ to be true). For WAND, the authors describe

Algorithm 3 The Weak-AND scoring algorithm.

```
WAND(Q, I, k, α)
 1  R = {}
 2  L_Σ = ∑_{q∈Q} LBnd(q, I)
 3  θ = −∞
 4  Q = PositionSort(Q, I)
 5  currentDoc = 0
 6  while TRUE
 7      pivot = FindPivot(Q, I, L_Σ, θ)
 8      if pivot == −1 or Q[pivot] has no more documents
 9          break
10      candidate = GetCandidate(Q[pivot], I)
11      if candidate ≤ currentDoc
12          // pick an iterator and move it past the currentDoc
13      else currentDoc = candidate
14          s_d = 0
15          foreach q ∈ Q
16              s_d = s_d + Score(currentDoc, q, I)
17          if s_d > θ
18              Insert(R, ⟨docid = d, score = s_d⟩)
19              if |R| > k
20                  Dequeue(R)
21                  θ = α · R.head.score
22  R′ = Reverse(R)
23  return R′

FindPivot(Q, I, L_Σ, θ)
1  s = L_Σ
2  i = 1
3  while s < θ and i ≤ |Q|
4      s = s + UBnd(Q[i], I) − LBnd(Q[i], I)
5      i = i + 1
6  if i > |Q|
7      i = −1
8  return i
```

the sentinels implicitly as a "pivot term", which is the first scorer in FindPivot such where $\sum_{q \in Q} \textsc{UBnd}(Q, I) > \theta$.

In both cases, only the scorers in the sentinel set provide candidates for scoring. The smaller the sentinel set is, the smaller the candidate pool is. If the end goal is scoring as few candidates as possible, then these algorithms attempt to limit the size of the sentinel set as a heuristic for approximating that goal.

2.3.2 Dynamic Optimization using Machine Learning

We now turn to two retrieval models that leverage the flexibility of machine learning in order to achieve high levels of efficiency.

2.3.2.1 Cascade Rank Model

The Cascade Rank Model (CRM) was developed by (Wang et al., 2011) as an efficiency approach for learning-to-rank (LTR) models, where the input for ranking is a vector of features. For each query, the features are extracted and are partitioned into sets of stages such that the features are ordered according to their time cost/rank performance. The authors then use a variant of the AdaRank algorithm (Xu & Li, 2007) to order the features into the stages. The intuition here is that instead of only learning the best ranking model, the algorithm learns the best ranking model given a certain execution cost.

The CRM is one of the few optimization techniques that can explicitly learn for budgeted time constraints. Given the ordering of stages, the CRM can use the partitioning to honor a possible time constraint parameter, such as one that might be given in a real-time system. After each stage, the algorithm can decide to continue

refining the running score of the current set of candidates, or terminate with the given candidate set. Note that given this construction, the CRM cannot guarantee score-safe ranking unless the time constraint is omitted.

Each stage S_i consists of a pair of components $\{J_i, H_i\}$, where J_i is a pruning function and H_i is a local ranking function. The correct composition and ordering of stages is what is learned via the AdaRank variant, which is shown in Algorithm 4.

Algorithm 4 The boosting algorithm for cascade learning.

LEARNCASCADE($\mathbf{Q}, \mathbf{J}, \mathbf{H}$)

1 $N = |Q|$
2 $\mathbf{S} = \{\}$
3 **for** each $q_i \in \mathbf{Q}$
4 $\quad P_1(q_i) = 1/N$
5 **for** $t = 1$ **to** T
6 $\quad$ Select $S_t = \langle J_t(\beta_t), H_t, \cdot \rangle$ over the training instances weighted by P_t
7 $\quad$ Set $\alpha_t = \frac{1}{2} \ln \frac{\sum_{q_i} \frac{P_t(q_i)}{1-\gamma \cdot C(S_t,q_i)} \cdot (1+E(S_t,q_i))}{\sum_{q_i} \frac{P_t(q_i)}{1-\gamma \cdot C(S_t,q_i)} \cdot (1-E(S_t,q_i))}$
8 $\quad$ Add $S_t = \langle J_t(\beta_t), H_t, \alpha_t \rangle$ to $\mathbf{S}$
9 $\quad$ Update P_t to P_{t+1}:
10 $\quad$ **for** each $q_i \in \mathbf{Q}$
11 $\quad\quad P_{t+1}(q_i) = \frac{\exp(-E(S_t,q_i)) \exp(\gamma \cdot C(\mathbf{S},q_i))}{\sum_{q_i} \exp(-E(S_t,q_i)) \exp(\gamma \cdot C(\mathbf{S},q_i))}$
12 **return S**

The parameters supplied to the algorithm are $\mathbf{Q}$, a set of training queries; $\mathbf{J}$, the set of pruning functions to select from; and $\mathbf{H}$, the set of local retrieval functions to cascade. We initialize the algorithm in Lines 1-4, particularly each training query is uniformly weighted. As we progress, at each iteration we construct the stage S_t that would provide the most improvement in retrieval effectiveness, balanced against the

cost of that stage (using the cost function C). We set the weight α_t for the stage, add it to the cascade, and update the weights of the queries (Lines 8-11). Note that in Line 6, function J_i is sampled from $\mathbf{J}$ with replacement, while H_i is sampled without replacement (otherwise we may construct a cascade of only a single ranking function).

2.3.2.2 Selective Pruning Strategy

The Selective Pruning Strategy (SPS) is a modification to the WAND algorithm by (Tonellotto et al., 2013). In this learned version of the algorithm, all query components (features) are given estimates towards their computational cost and their difficulty. Using these estimates, the number of requested results (K) and the aggressiveness (α) of WAND are estimated and used to run the WAND algorithm for an initial retrieval. If a learning component (such as AdaRank) is also part of the model, then the learned model features are extracted from the top-K results, and used for re-ranking the top-K with the learned model. This is not unlike the CRM, in that low-cost evaluation is performed earlier in the retrieval, and high-cost evaluation is used to refine the initial result list.

Algorithm 5 The Weak-AND scoring algorithm.

SelectivePruning($\mathbf{Q}$)
1 $\{K, \alpha\}$ = Select(Predict(Q))
2 S = Wand(Q, K, α)
3 F = Extract(Q, S)
4 R = Apply(F, Model(K, α))
5 **return** R

Both processes can be paralleled with standard result refinement or reranking approaches. The CRM begins with an approximate ranking, but uses additional information external to the initial retrieval to iteratively refine the results. SPS more closely uses the base model as a fast approximation of the end results, and then refines its result using a learned model where the expected execution cost is much higher.

Another noteworthy aspect of both approaches is that they fundamentally rely on the same mechanism for optimization as the algorithmic methods: the way in which the query is scored must be decomposable in some way. Specifically, all four algorithms receive some set of components that when taken as a whole (e.g. summed together) represent the entire evaluation of the query. We shall leverage this fact later on to show that the optimizations provided here can transfer from the algorithmic techniques to the machine-learned ones.

2.4 Query Languages in Information Retrieval

In order to focus optimization efforts, we must first determine what kinds of operations we should expect to encounter in a given query. This is determined by the elements available in the query langauge of the search system. Towards this end, we now briefly review the types of operations that have come to be supported in contemporary commercial and open source search engines.

2.4.1 Commercial Search Engines

Early uses of search in industry focused on Boolean-style retrieval, where documents were returned only if they fit the query which was specified using a set of terms and Boolean operators (Sanderson & Croft, 2012). Considering these early systems, completely structured queries were the norm in the early days of commercial search. As research in IR advanced, it became clear that ranked (or "best-match") retrieval regularly produced better results than Boolean retrieval. In the 1990s, commercial systems moved towards ranked search, which gained momentum at this time with the proliferation of web search engines and the dot-com bubble. Queries in ranked retrieval did not require the specification of a Boolean-logic query, and therefore provided a more intuitive interface to non-expert users. Therefore this change also meant that explicit structure in queries became the exception instead of the norm. Ironically, even though web search engines encouraged users to input queries with little to no structure, these companies have come to spend a significant amount of effort inferring the structural components in the queries that rarely express structure explicitly.

An informal survey of global market share indicates that the Google, Bing, and Yahoo! search engines carry well over 90% of the internet search market (StatCounter, 2011). Given the majority wielded by these sites, we assume that query language elements featured in these sites are representative of common elements in modern commercial IR systems. An investigation of the Advanced Search features of these sites reveals similar capabilities among the three systems, including the following: *unigrams*; *exact phrase matching*; *synonyms*; *Boolean operations* (AND/OR of the

given words, NOT this[2]); *field comparison* (reading level of X or higher..., language, file format, date filtering, URL filtering); and *field-aware retrieval* (match query within a field).

2.4.2 Open Source Search Engines

The evolution of open source search engines reflects the gravitation of IR research towards structured queries as well.

One of the earliest open source systems in modern retrieval is the SMART retrieval system (Salton, 1971). SMART used the Vector Space Model (Salton, Wong, & Yang, 1975), and served as the proving ground for a significant portion of the community's understanding of ranked retrieval. While groundbreaking in many ways, the input interface to SMART is relatively simplistic, and queries can only be entered as simple strings. This kind of interface was standard for information retrieval systems through the 1980s.

The next major advancement came from the INQUERY retrieval system (Callan, Croft, & Harding, 1992). INQUERY was built to implement the Inference Network framework (Turtle & Croft, 1990), a subset of the *graphical models* domain of reasoning (Manning, Raghavan, & Schütze, 2008; Koller & Friedman, 2009). INQUERY supplied an expressive query language that allowed for query operators to be used on base query terms. Additionally, some operators could be applied over other operators, allowing for query construction using hierarchical element structure. As

---

[2] Note that NOT may only apply to a conjunction (i.e. AND (NOT x)). The disjunctive form (i.e. OR (NOT x)) is not informative for boolean query processing.

mentioned earlier, expressing queries with structure was not new, but this represents the first case of structural queries being used in a ranked retrieval setting. The effects of joining these two strategies can be seen in many modern day commercial and open source systems.

The Indri retrieval system (Strohman, Metzler, Turtle, & Croft, 2005) was built as a system that used Inference Network (and the query language coincident with it) to implement Language Models (Ponte & Croft, 1998). Indri added several new operators to the base set of INQUERY - while this increased what kinds of belief estimations could be made in Language Models, it did not significantly change the nature of query expression as INQUERY had.

The Galago (Strohman, 2007; Croft, Metzler, & Strohman, 2010) retrieval system serves as an evolutionary step past systems built to statically support a single retrieval model. Galago is not tied to a specific retrieval model, and instead can support arbitrary functions built on term-,document-, and collection-level statistics supplied from the index. The system ships with several retrieval models implemented, and provides several mechanisms for adding new operators. In a sense, this means Galago does not have a bounded query language, which makes it the most expressive retrieval system known to date.

Several other open source systems have been built to provide similar structured query elements as the original INQUERY system. Zettair is the most straightforward system, allowing for use of unigrams and phrase searches in ranked and Boolean queries (Zobel, Williams, Scholer, Yiannis, & Hein, 2004). A brief view of the Terrier (University of Glasgow, 2011) retrieval system's query language shows the use of

similar constructs: synonyms, ordered and unordered windows, fields, and Boolean search. The Wumpus system makes use of generalized concordance lists (GCLs) (Clarke, Cormack, & Burkowski, 1995), which allow for interesting interactions between extents of text in documents, but do not introduce any new widely adopted operations. Most of the operations enabled by GCLs are directly expressible in the INQUERY query language. The Ivory system[3] implements the SMRF framework (Search with Markov Random Fields) (Lin, Metzler, Elsayed, & Wang, 2010). The SMRF framework was first implemented informally in the Indri search engine (Metzler & Croft, 2005), using the query language built in with the Indri. Therefore we consider the query language no more expressive (and possibly a restriction of) Indri.

One of the newest open source search engines is ATIRE, built at the University of Otago[4]. In an odd reversal of the trend, ATIRE was built without support for term dependencies (i.e., phrases or window operators), as the implementing group has not been convinced that supporting term dependencies is worth the increased complexity in retrieval inference (Trotman, 2012). This point of view shows that even 20 years after the first INQUERY implementation, building a search engine to support structure in ranked retrieval is still not a trivial matter (although it is worthy to note that commercial search engines have committed to supporting several types of structure, including phrases).

---

[3] `http://lintool.github.io/Ivory/`

[4] `http://atire.org/index.php?title=Main_Page`

In addition to the academic systems mentioned above, several other open source implementations exist, in many cases as alternatives for "single-site" search installations. These systems tend to be geared for commercial-scale tasks, but are available for use by anyone willing to set the system up. The most well-known of these is the Apache Lucene Core system[5]. Lucene began in 1999 by Doug Cutting as an exploration of implementing a retrieval system in Java. Over time, the project was adopted by the Apache Foundation, and over several iterations, Lucene grew to focus on meeting the needs of specific clients. As a result, the query language of Lucene covers the functionality offered by the commercial search systems, as well as several extensions, including *edit distance matching*, *ranged Boolean queries on fields* and *unigram wildcard matches* (Apache Software Foundation, 2012).

If we compare the query language elements available in each of the systems reviewed, two tiers of "complexity" emerge: the first is support for just specifying terms, as in SMART, Zettair, or ATIRE. These systems can provide good support for simple keyword retrieval models, however expressing a more sophisticated model may be difficult without major implementation changes. The second tier is defined largely by INQUERY: systems in this tier support structured query constructs, allowing them to express a much wider range of queries than the simpler systems. Although Galago technically sits above this tier due to the flexibility of the query language, the tier above the current one is unclear, as no significantly new retrieval models have been implemented in the system (possibly due to a lack of retrieval mod-

[5]`http://lucene.apache.org`

els that clearly "out-express" the one put forth by INQUERY). A formal exploration of new types of queries is beyond the scope of this thesis, we discuss possibilities for research in Section 7.2. Consequently, we place Galago in the second tier, along with other systems which sport structured queries.

2.4.3 Important Constructs

After reviewing these systems, we determine that the following query constructs are important:

Ordered {n,k}-Grams A user can specify n grams[6], to appear in a strict total order in the order the grams appear, with no more than k grams between each pair of sequential grams. Phrases of length n are a special case of {n,0}-Grams.

Unordered{n,w}-Grams n grams appear within a window size of w grams, in any order.

Boolean Filtering Documents must pass a Boolean test before being scored. Note that Boolean retrieval is a special case of this operation.

Metadata Mapping We allow mapping functions of the form $f : d \in D \rightarrow \Sigma$, where D is the universe of documents in the collection, and Σ is some target universe of symbols. f is required to be an injective function. An example of this operation would be mapping every document to the date it was published.

[6] We define a gram as a single token or term.

Extent Storage We allow the storage of extents – windows of tokens under some label. This can be viewed as a generalization of Metadata Mapping, except here the possible values in each field can be whole sequences (i.e. substrings) of tokens from the vocabulary of each document. These form the basis of the field-based retrieval models, which we will discuss in Section 4.2

Although other constructs are certainly possible, we start with a focus on the ones presented above; they already have wide acceptance and support in the majority of retrieval system implementations, and a large body of research has shown the effectiveness of each construct.

2.4.4 Other Areas of Optimization

Information Retrieval is certainly not the only discipline to be active in efficiency research. In particular, if we were to look at optimization literature in such fields as compiler and database research, it is clear that research concerning efficiency is slightly more mature than in IR, and researchers approach optimization at multiple levels of resolution. We now discuss these two areas, highlighting their similarities and differences to the state of IR optimization research, and in particular work discussed in this thesis.

2.4.4.1 Compiler Optimization

In compiler optimization, the approaches are usually classified according to scope. For example, optimizations such as function inlining may operate at the global level, after all code has been brought together and converted into some intermediate representation. In contrast, a peephole optimization, such as a strength reduction,

only operates on a handful of instructions at a time. Other optimizations, such as loop unrolling, only operate on code involved in the execution of loops, which fit in between the prior two scopes. In more recent years compiler optimization has grown to involve a large amount of flow control and dependency analysis (Allen & Kennedy, 2001), in particular to leverage advances in CPU instruction handling (e.g. pipelining, vectorization, and branch prediction). The advent of multi-core CPU architectures has also added motivation for compilers to handle parallel operations in modern day programming languages.

While compiler optimizations clearly have an impact on the performance of any program, the approaches taken by the compiler community do not readily transfer over to information retrieval, as the domains are too different to reduce one problem to a similar one in the other domain.

2.4.4.2 Database Optimization

Database optimizations, at the highest level, are usually classified as either *physical optimizations*, which focus on improving low-level operations in a database, such as the time it takes to perform an index lookup, and *logical optimizations*, most of which involve reorganizing the sequence of operations necessary to fulfill the query. This sequence is usually called the *query plan*, and the subsystem responsible or analyzing and logically optimizing the plan is referred to as the *query optimizer*.

The idea of a query optimizer subsystem was discussed in the early seventies (Wong & Youssefi, 1976; Selinger, Astrahan, Chamberlin, Lorie, & Price, 1979). Most query optimizers represent the query plan as a tree, where a node in the tree represents a single operation in the query. Therefore, one can simply convert a query

in relational algebra into a query tree that the optimizer can then operate on. The tree itself represents a state, and each optimization is an action on that tree; taken in this way, the problem fits nicely into a search problem, where the optimizer is searching for the minimal cost tree to run against the data. Most implementations assume that execution of the query will take far longer than the optimal tree search, therefore the search is run exhaustively over the space of possible trees. This process is most useful when a query contains multiple table joins, and the order of selection, projection, and joining of tables can change the execution time of queries by orders of magnitude. This analysis is commonly referred to as *join reordering* in the database literature. Note that often, join reordering is augmented by *cost estimation*, where the database stores estimated costs for reading a table in a certain way (a table scan costs a lot more than an index scan, for example). In addition, the optimizer leverages *dynamic programming* to cache completed calculations of subtables during execution, and *interesting orders* to analyze the output sort order of calculated subtables, in case sorts can be removed due to redundancy (Chaudhuri, 1998).

Optimization in Information Retrieval lacks formalities such as query plans and query optimizers, arguably because the mathematics involved in retrieval models is not rigidly defined, whereas database systems are fundamentally based to service queries formed in relational algebra. The techniques described above have all been designed to operate on relational algebra of typed and highly structured data, but the data operated on in IR has only been assumed to be text documents, with little well-defined structure (handling unstructured or semi-structured data has been one of the defining attributes in discerning IR from databases). Due to this fluidity, the

majority of optimization research in IR would most likely be classified as physical optimizations, whereas less progress has been made at the logical level.

All retrieval systems to date can benefit from improvements in posting list traversal. Such improvements include compression algorithms with better (de)compression speeds and factors, integrated skip list structures, or different organizations of the posting list information that allow for more fine-grained decoding of postings. In the context of this thesis, we consider these improvements to be *physical* optimizations, as they are applied below the context of interest, and the implementation calls where these optimizations take place are treated as atomic operations to the algorithms higher up the call stack. The optimizations of interest we consider to be *logical* optimizations, as we analyze and compare the actual algorithms at this point.

2.4.4.3 Ranked Retrieval in Databases

Of special interest are the "top-k" retrieval algorithms. The extension of database retrieval to a "ranked" system comes from the top-k sub-community, with the introduction of top-k retrieval by Fagin (Fagin, 1996). In this seminal paper, Fagin describes a straightforward algorithm for returning an ordered set of results, based on some scoring metric. If we view posting lists as single-column tables, the mapping between databases and IR systems seems painless. However there are some important differences between the two domains. Most notably, Fagin assumes that all implementing systems support random-access operations, even if they cost more than a forward scan in the same table. One of the reasons the IR community developed their own data structures was to avoid the need to support random-access lookups. By supporting only forward-scan operations, IR systems were able to quickly outperform

similar systems built on top of traditional DB architectures. While random-access support could be added to an IR system (simply reset and forward scan to the requested location), it would undermine a significant amount of progress made based on the forward-only assumption.

A noteworthy extension to relational databases are *probabilistic databases* (Dalvi & Suciu, 2007). A probabilistic database uses *possible worlds semantics*, where the rows stored in the database allow for distributions over the values in a given column. Although probabilistic databases are often paralleled to ranked IR systems, they are on opposite ends of a spectrum. As Dalvi and Suciu stated, several query constructions in probabilistic databases are #P-complete, with complexity worsening as the number of uncertain predicates increases. Conversely, ranked IR commonly works with only uncertain predicates, with various extensions that allow for set-like matching and filtering (e.g. boolean modifiers or temporal restriction on results). Considering this disparity in assumptions and common use cases, probabilistic databases, and optimizations developed for such systems, are not considered here.

Now that we have established the landscape of optimization in information retrieval and related fields, we now introduce the first of the three optimizations discussed in this thesis: simplifying the structure of queries to improve their execution performance.

CHAPTER 3

FLATTENING QUERY STRUCTURE

Researchers in IR have recognized the utility of structured queries in ranked retrieval for over 20 years (Turtle & Croft, 1990; Callan et al., 1992). In the interim, research in IR has continued to produce evidence supporting this idea (Metzler & Croft, 2005; Bendersky & Croft, 2008; Lavrenko & Croft, 2001; Bendersky & Croft, 2008; S. Robertson, Zaragoza, & Taylor, 2004; Yan et al., 2010; Huston et al., 2011). In addition, various evaluation conferences have led efforts towards researching the use of structure in queries, such as INEX[1] and TREC's Entity Track[2]. Assuming that queries have some structure has become the expectation in retrieval; unfortunately large complex queries are likely to be slow to process, indicating a clear need for optimization if such queries have any hope of being called interactievly. Towards this end, we look to improve query processing on a recurring pattern in information retrieval models: linear combinations.

Linear combinations of scores lie at the heart of many well-known models in information retrieval. The class of probabilistic models contains, or easily translates to, linear combinations of unigrams or n-grams. The Language Model (Ponte

[1] http://www.inex.otago.ac.nz/

[2] http://ilps.science.uva.nl/trec-entity/

& Croft, 1998) is technically a ranking based on joint probability of the query terms $q_1 q_2 \ldots q_k = Q$ occurring in document D; however in order to avoid underflow the scores produced are the sum of the log-probabilities: $Score(Q, D) = \sum_{q \in Q} \log P(q|D)$. A term-dependency aware extension to the Language Model, the Sequential Dependence Model, uses the weighted geometric mean to combine the unigram, ordered, and unordered components of a given query (Metzler & Croft, 2005). When we take the logarithm of these quantities, the entire equation again reduces to a series of sums over log-probabilities. The Relevance Model (Lavrenko & Croft, 2001), after generating the estimates for the likelihood of a term w coming from relevance model R, denoted $P(w|R)$, is a linear combination of log-probabilities as well: $\sum_{w \in V} P(w|R) \log P(w|D)$. Likewise, the binary independence models, most famously the BM25 model (S. E. Robertson & Walker, 1994), are also linear combinations of term weights.

If we look to vector space models, we can observe a similar phenomenon. The standard cosine similarity scoring function takes the inner product of D and Q — which in turn is a sum of products between the two vectors (Salton et al., 1975). The Rocchio algorithm for relevance feedback (Rocchio, 1971) creates an interpolation between the components of each query term:

$$q'_j = \alpha \cdot q_j + \beta \cdot \frac{1}{\|R\|} \sum_{D_i \in R} d_{ij} - \gamma \cdot \frac{1}{\|\bar{R}\|} \sum_{D_i \in \bar{R}} d_{ij}$$

These new weights are then used to provide a "more complete" vector representation of the information need. In implementation, the score consists of calculating the dot product, as before. There are numerous other examples of established models that

operate as linear combinations of various components, and linear combination is often one of the first methods tried when adding new information into existing models.

In this chapter we work to improve the effectiveness of the two algorithmic optimizations (Maxscore and WAND) with respect to the class of queries that are formulated not only as linear combinations, but as *nested* linear combinations. In this context we call a linear combination present in the query an *interpolated subquery*. By identifying and removing as many interpolated subqueries as we can prior to query evaluation, we can dramatically improve pruning effectiveness, resulting in substantial increases in evaluation speed.

We perform three studies in this chapter. For the initial study, we limit the number of subqueries, but we vary the way in which we can interact with them. This limits the number of confounding factors that may interfere with the outcomes, and the resulting analysis illustrates the relationship between the flexibility of a query's structure and query execution time. The second study is motivated by the results of the first, and we look closer at the correlation between the increase in exposure of a query, and the corresponding increase in execution speed. Finally, we examine the effect of ordering the scoring elements by impact instead of by the length of the posting list. Under certain circumstances, we can improve performance even further. We empirically show the effect of these treatments on the AQUAINT and GOV2 document collections, using queries from the TREC Robust 2006 and the Terabyte Efficiency tracks. Most experiments presented in this chapter were originally conducted using the Galago v3.3 retrieval system - supplemental experiments were conducted using the Julien retrieval system, and we note where this is the case.

3.1 Interpolated Queries

In the query tree model, a simple (i.e., disjunctive) keyword (KEY) query is any query that can be represented as a "flat tree", as shown in Figure 3.1. If we take a probabilistic interpretation to each node, then the model can be mathematically equated to a Naïve Bayes Classifier, where each input feature is independent of the others. Maxscore and WAND were both designed to work over this class of queries, and do so effectively.

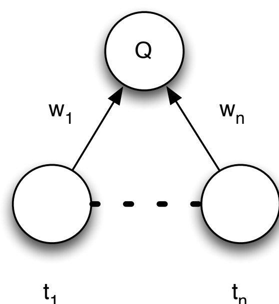

Figure 3.1. A weighted disjunctive keyword query represented as a query tree.

However, the KEY representation does not cover the whole space of known effective retrieval models. Consider the query `'new york' baseball teams`. Unless the phrase `'new york'` is stored as a unique posting list in the index, the phrase must be generated at query time by intersecting the `new` and `york` posting lists. However, this also makes the query tree no longer "flat", as shown in Figure 3.2.

This example introduces additional hierarchy which extends beyond the single-layer tree representation. The root node of the tree no longer has leaves as children; each node may now be its own non-unigram subquery, such as a phrase or a set

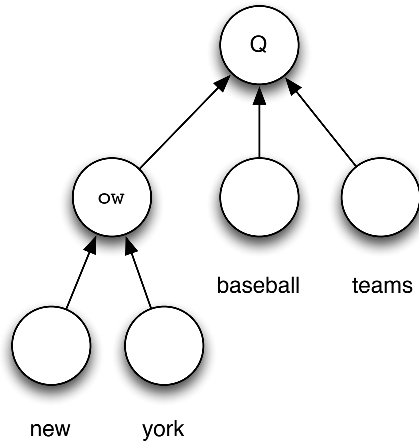

Figure 3.2. A “non-flat” query tree, representing the query `‘new york’ baseball teams`.

of expansion terms. It is this class of query we call an interpolated query, as the weights assigned to the subquery nodes may be normalized to be an interpolation of the subqueries. Viewed in the query tree representation as in Figure 3.3, the top scoring node contains any number of interior subquery nodes, and the final score for a document consists of the scores of these nodes being linearly combined.

Clearly, the set of interpolated subqueries forms a superset of queries that can be expressed using KEY, and the research of the IR community reflects this fact. Yet, before this study, no one had investigated the effects of such increased structure on query performance. We now move on to the first study conducted in this chapter.

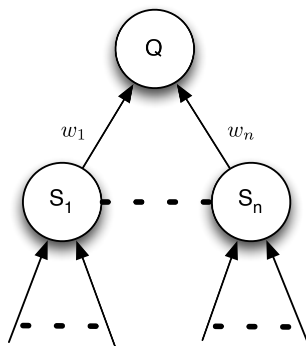

Figure 3.3. The general form of an interpolated subquery tree. Each subquery node S_i may be a simple unigram, or it may be a more complex query tree root at S_i.

3.2 Study 1: Modifying a Single Interpolated Subquery

The basic structure we consider is a two-part query, which we first formulate as an equation:

$$Score(Q, d) = \lambda F_1(w_1^u u_1 \dots w_n^u u_n) + (1 - \lambda) F_2(w_1^v v_1 \dots w_m^v v_m) \tag{3.1}$$

where $Score(Q, d)$ is the desired score for document d with respect to query Q. λ is a tunable parameter (or a set weight), and F_1 and F_2 are functions over d and the provided terms. The weight arguments (denoted as w) apply to the term of the same index in that function. The sets of terms $\{u_1 \dots u_n\}$ and $\{v_1 \dots v_m\}$ may include terms from Q, however this is not a requirement. To better explain the interpolated query structure, consider the following example query *hydrogen energy*, augmented with several expansion terms:

$$Score(\texttt{hydrogen energy}, d) = \lambda F_1(w_1^u \texttt{ hydrogen } w_2^u \texttt{ energy}) + (1-\lambda)F_2(w_1^v \texttt{ science } w_2^v \texttt{ nuclear } w_3^v \texttt{ research }) \quad (3.2)$$

We consider the linear interpolation structure in four different scenarios, shown in Figure 3.4. We show actual terms connected to the nodes, while the associated weight is shown along the edge from Equation 3.2. When a node is shaded, it means that the internal structure of the node cannot be manipulated - from an operational perspective it is a fixed "black-box". In other words, either we have no access to the internals of the node itself (e.g. a node processed remotely), or if we were to modify the structure of the subtree under this node, the resulting query tree would be semantically different from the original one. In this case we say the node is "immutable", since its structure cannot be mutated. An unshaded node is therefore "mutable", meaning that we can access, and if necessary, restructure the child nodes in order to improve execution at query time.

Scenario 1 of Figure 3.4 is the case where both subqueries are considered immutable. This case occurs any time a component is an n-gram operator; if we restructure the subtree rooted at that operator, we change the semantics of the query. This case may also occur when we receive the components from two different servers (which may occur in a distributed index setting), or more generally, if the components point to two different types of data (e.g., a static rank function and a query-dependent function). In Equation 3.2 above, this may occur if F_1 is a phrase operator over the terms, and F_2 defines a window over the provided terms. Neither

function can be manipulated without affecting their semantics. In Scenarios 2 and 3, one of the nodes is immutable, and another is mutable. This can be achieved when either F_1 or F_2 maps to a disjunction operator. Finally, in Scenario 4, both nodes are mutable and can be restructured. This occurs if both F_1 and F_2 are disjunction operators.

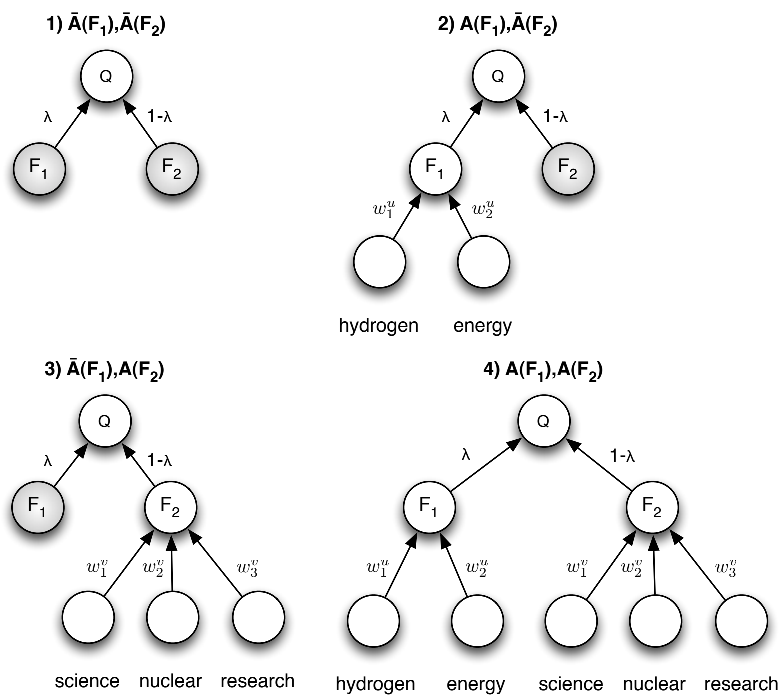

Figure 3.4. Four cases under investigation, with varying amounts of mutability. Shaded nodes are immutable.

3.2.1 Experiments

We select the RM3 variation of the Relevance Model as the test retrieval model (Abdul-jaleel et al., 2004). RM3 naturally forms a linear interpolation structure:

$$Score_{RM3}(Q,D) = \lambda Score_{LM}(Q,D) + (1-\lambda)Score_{RM}(Q,D)$$

where $F_1 = LM$ and $F_2 = RM$. To test the sensitivity of the algoithms to the tree structure, we modify the tree according to the cases outlined above. Case 1 is the baseline model, where we do not modify the tree at all. In Case 2, we modify the subtree under F_1. To better expose that subtree, we move the leaf nodes directly under the root, allowing it to prune after each term in the subtree. Graphically, this operation is shown in Figure 3.5.

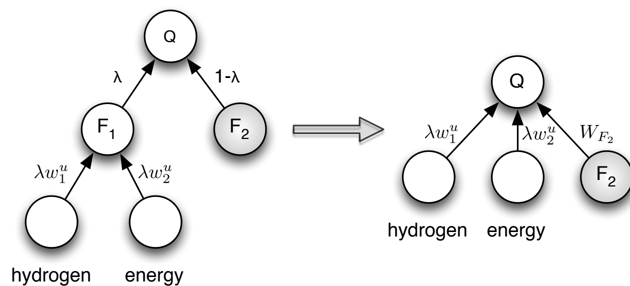

Figure 3.5. Reducing the depth of a tree with one mutable subquery node.

The redistribution of weight among the nodes under F_1 is dependent on the model itself. In the case of interpolated subqueries, λ acts a multiplicative factor applied after summing the score contributions under F_1. If we substitute the implementation of F_1 and F_2 into Eq 3.1, we get:

$$Score(Q,d) = \lambda \sum_{i=1}^{n} w_i^u u_i + (1-\lambda) \sum_{j=1}^{m} w_j^v v_j \tag{3.3}$$

From a mathematical perspective, flattening out F_1 is the equivalent of distributing λ among the inner terms of the sum and making the inner terms part of the outer summation:

$$\begin{aligned} Score(Q,d) &= \lambda \sum_{i=1}^{n} w_i^u u_i + (1-\lambda) \sum_{j=1}^{m} w_j^v v_j \\ &\rightarrow \lambda w_1^u u_1 + \cdots + \lambda w_i^u u_i + \cdots + \lambda w_n^u u_n + (1-\lambda) \sum_{j=1}^{m} w_j^v v_j \end{aligned}$$

The post-operation tree we label as MAX-F$_1$ in the experiments. In the reverse case (Case 3), we perform the mirror of this operation, and call the resulting tree MAX-F$_2$. Note that MAX-F$_1$ and MAX-F$_2$ are isomorphic; we label them separately for the sake of clarity in the results. In the experiments F_1 forms the Language Model part of the query, while F_2 forms the Relevance Model part. We vary the length of the expansion terms used in RM while the number of terms in the original query is held constant.

Case 4 allows access to both subtrees, therefore we flatten both subtrees out, returning to a flat tree version of the RM3 query. The flattening operation is shown in Figure 3.6. In the experiments, this tree is labeled MAX-FLAT.

We vary three parameters when performing experiments:

Interpolation Weight The weight λ determines the importance of each subquery. We investigate three distinct values of λ: 0.2, 0.5, and 0.8 to examine how changing the importance of the subqueries affects each of the algorithms.

Number of Expansion Terms The ratio of terms between F_1 and F_2 will most likely impact the effectiveness of the algorithms. Let $\tau = |\hat{F}_2|$, the number of

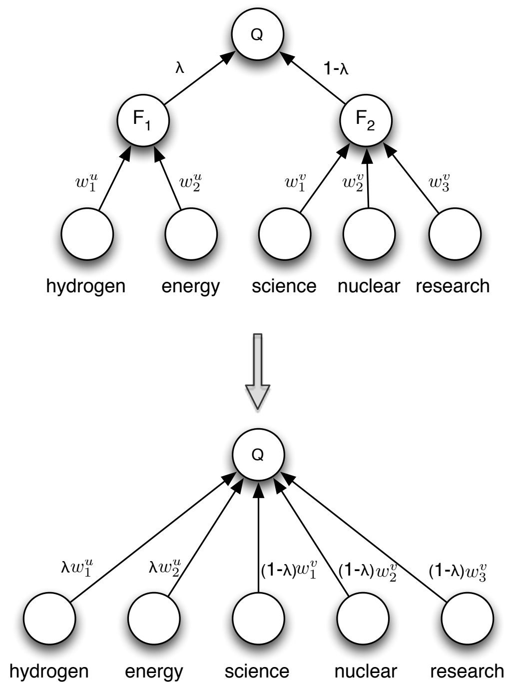

Figure 3.6. Completely flattening the query tree.

terms in the expansion. We use values of 10 and 100 for τ to examine minor and major imbalance. Terms from the original query are filtered from the set of expansion terms, to ensure that the two sets are disjoint.

Number of Items Requested We vary r, the number of documents requested in R, to determine how the algorithms are affected by needing to maintain larger candidate lists. We use values of 100 and 1000 to observe this effect.

For each parameter setting, we provide the average number of times the Score function was called, and the percent change from the average for the RM3 method. In all cases, the *default* setting indicates the following parameter values: $r = 100$,

$\lambda = 0.5$, and $\tau = 10$. We include an RM3 run that makes use of Maxscore (RM3+), for comparison. Table 3.1 contains the experimental results when tracking the number of Score calls. Each row provides the results for a particular parameter setting in a given collection, and a column is the performance of a method over the different parameter settings.

Tracking Score calls is informative, however it may mask overhead incurred when deciding whether to evaluate a scorer or not. To 1) examine the real effects of the optimization and 2) examine the accuracy of Score counting, we also examine changes in performance based on system time measurements. We perform 5 runs of the GOV2 collection, with the queries permuted randomly in each run to lessen any dependence on query order. We record the time to execute each query for each method, making a total of 750 samples for each algorithm. To compute statistical significance between algorithms, we use a randomization test of 1 million samples with the sample mean as the sufficient statistic. We also determine what percentage of the queries shows some "positive effect" relative to unmodified RM3. Let $\mathcal{A}$ be the algorithm in question. If the value for $\mathcal{A}$ is less than 90% of the value for RM3 for a particular measure (either score count or real time), we consider that to be a positive effect. For example, a value of 10 in the table indicates that 10% of the queries (75 of 750) had that measure reduced by at least 10% compared to RM3. Table 3.2 shows the mean runtimes of the samples, the percent change from RM3, and percentage of queries affected when measuring via score count and real time.

3.2.2 Results Analysis

We now turn to experimental results (Cartright & Allan, 2011) which provide an evaluation of these four scenarios[3]. The results from the three datasets look largely equivalent, suggesting that the methods described here will retain their ability to improve over the baseline as the collection size increases. Both RM3+ and MaxPlus seem to generally perform better on the Clueweb-B dataset; analysis of this phenomenon reveals that the proportion of queries actively optimized (i.e. pruned) is much higher in Clueweb-B ($> 45\%$ in both cases) than in the other two collections ($< 23\%$). This in turn produces a larger impact on the average SCORE count.

Changes in performance based on system time measurements are presented in Table 3.2. What percentage of the queries shows some "positive effect" relative to unmodified RM3 is also shown. Let $\mathcal{A}$ be the algorithm in question. If the value for $\mathcal{A}$ is less than 90% of the value for RM3 for a particular measure (either SCORE count or real time), we consider that to be a positive effect. For example, a value of 10 in the table indicates that 10% of the queries (75 of 750) had that measure reduced by at least 10% compared to RM3. Table 3.2 shows the mean runtimes of the samples, the percent change from RM3, and percentage of queries affected when measuring via SCORE count and real time.

Based on the results from Table 3.2, the algorithms seem to fall into 2 distinct groups. The strongest of these is MAX-FLAT, and to a slightly lesser degree, MAX-F_2, which seem to consistently reduce both the SCORE count and the system time by

[3] We relabel the different scenarios as needed, and only show results pertinent to the current discussion.

Aquaint	RM3	RM3+	Max-F_1	Max-F_2	Max-Flat
Default	8.5	-4.4†	-7.1†	-68.3	-85.3
$r = 1000$	8.5	-4.4†	-7.1†	-64.6	-79.1
$\tau = 100$	104.8	-1.6†	-1.5†	-46.8	-61.9
$\lambda = 0.2$	8.5	-3.0†	-2.7†	-73.1	-80.0
$\lambda = 0.8$	8.5	-86.5	-92.3	-69.4	-87.6
Gov2	RM3	RM3+	Max-F_1	Max-F_2	Max-Flat
Default	193.2	-7.2	-9.0	-56.5	-88.2
$r = 1000$	193.2	-0.6†	-9.0	-53.4	-82.1
$\tau = 100$	2351.7	-2.8	-2.9	-50.8	-75.2
$\lambda = 0.2$	193.2	-2.4	-1.5	-74.7	-78.0
$\lambda = 0.8$	193.2	-86.7	-94.2	-54.5	-87.6
ClueB	RM3	RM3+	Max-F_1	Max-F_2	Max-Flat
Default	400.4	-23.7	-27.1	-58.0	-90.7
$r = 1000$	400.4	-23.7	-27.1	-57.0	-90.7
$\tau = 100$	4428.7	-11.6	-14.8	-58.3	-81.2
$\lambda = 0.2$	400.4	-13.3	-8.5	-81.2	-87.6
$\lambda = 0.8$	400.4	-68.9	-90.6	-58.1	-93.6

Table 3.1. Results for the Galago retrieval system (v3.3) over AQUAINT, GOV2, and ClueWeb-B, using 36, 150, and 50 queries, respectively. The number in the RM3 column is the number of score requests (in millions) using the unmodified algorithm. The numbers in the remaining columns are the percent change relative to the the unmodified RM3 model. We calculate this as $(B - A)/A$, where A is RM3 and B is the algorithm in question. The † indicates a change that is *not* statistically significant.

a significant amount. Their average impact is high, and they cover a high percentage of the queries. The second group consists of RM3+, and Max-F_1. The average runtimes are greater than the unmodified RM3, but they still register at least a 10% measurement improvement for some number of the queries tested. Further analysis of this last group shows that when the algorithms work, they have a significant impact on both measurements, often resulting in reductions over 40%. However, as

the table shows, these algorithms do not "trigger" very often, and the added logic used to continually check for a pruning opportunity results in a notable increase in execution time.

Method	Mean Time	Pct Chg	Pct. Affected	
			Score	Time
RM3	134.8	0		
Max-F_2	68.6	-49.1✦	87.3	86.1
Max-Flat	19.4	-85.6✦	100.0	100.0
Max-F_1	230.7	+71.1	22.0	13.7
RM3+	245.9	+82.4	12.0	7.1

Table 3.2. Statistics over 750 queries run over GOV2. Mean times are in seconds. The ✦ indicates statistical significance at $p \leq 0.02$. The Score and Time columns report the percentage of queries that experienced at least a 10% drop in the given measurement.

We also conducted a similar study for the Wand scoring regime, using similar parameter settings as in Table 3.1, but reporting on percent change in wall-clock time instead of score count. The results are shown in Table 3.3.

Aquaint	RM3	RM3+	WAND-F_1	WAND-F_2	WAND-Flat
Default	964.6	-49.6	-72.6	-2.0	-49.5
$r = 1000$	966.0	-41.7	-42.9	0.8	-16.8
$\tau = 100$	12997.6	-66.9	-73.8	163.1	99.5
$\lambda = 0.2$	953.6	-48.9	-77.4	-2.0	-51.4
$\lambda = 0.8$	953.2	-47.0	-51.5	-1.0	-38.7

Table 3.3. Wall-clock time results for the 4 configurations scored using Wand over the Aquaint collection. Experiments conducted using the Julien retrieval system.

The results also show significant improvements, but in a different way. WAND-F_1 dominates the other configurations, and surprisingly, the WAND version of RM3+ performs nearly as well. WAND-F_2 shows little improvement over the original base

run, except in the $\tau = 100$ configuration, where the performance is considerably worse than doing nothing. The WAND-Flat configuration performs reasonably well in most configurations, but has terrible performance in the $\tau = 100$ configuration.

3.3 Study 2: Examining the Correlation Between Exposure and Speedup

The first study in this chapter indicates that the more of the query we can directly expose to the pruning logic of the optimizations, the more chances the algorithms have to elide work. Based on this lead, we now look at the correlation of what percentage of the query we are now exposing, and the resulting effectiveness of the optimization algorithms.

3.3.1 Experimental Setup

The software and hardware configurations are similar as before, but we now focus solely on establishing an explanatory variable and response variable correlation.

Note that although we would like to increase the size of the query under the root, the depth of the tree is not as important a factor as simply the number of computations hidden under the root. Consider the two query constructions in Figure 3.7. Although the upper query is deeper than the lower one, both queries provide the same opportunities for pruning in their original forms. The scoring nodes used in pruning are indicated by the triangular nodes. If we assume that all of the non-leaf nodes in the query meet our criteria for flattening, then the flattening process will

reduce both queries to only make use of the leaf nodes, indicated by the square nodes in the figure.

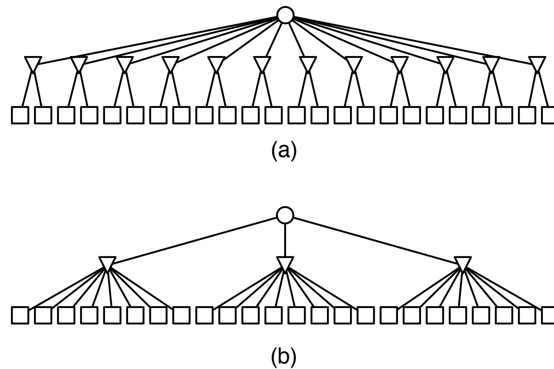

Figure 3.7. Comparison of a deep query tree, vs a wide tree with the same number of scoring leaves. To the dynamic optimization algorithms of today, the two offer the same chances for eliding work.

In order to generate an appropriate query set for this experiment, we take the following steps:

1. We intersect the vocabulary of the collection in question against the common American dictionary words on the Ubuntu 12.04 Linux distribution [4].

2. Shuffle the remaining set of words using the `Random` class provided by Java 1.6 using a seed value of 100.

3. We then iteratively generate random numbers between 2 and 50, using the `Random.nextInt` function, until the sum of these numbers is greater than the

[4]The file used is `/usr/share/dict/american-english`.

size of the vocabulary. We then partition the sequence of words according to these numbers, using them as lengths for the resulting subsequences of words (i.e. if the first three numbers are 3, 17, and 20, we would take the first 3 words to make the first query, then the next 17 words for the second query, 20 words for the third query, and so on). The last query is either the number of remaining words, or the size of the last number generated.

Using the steps above, 2280 queries were generated. For each query we then generate up to 50 feedback terms using Relevance Model-driven PRF. Let $Q = q_1 \ldots q_n$ be a query, and $E = e_1 \ldots e_m$ be a set of feedback terms. Keeping in mind that we want to explore the effect of exposing more nodes for pruning via flattening, we generate a particular query to as follows. Let b be the branching factor, and let $l = \lfloor(|Q|+|E|)/2\rfloor$. We vary b between 2 and l, and for a given setting of b, that is the number of nodes under an interior node in the query tree. Therefore, if $|Q|+|E| = 20$ and $b = 3$, then the tree will have a root node with 6 children nodes, each of which will have 3 leaves under it (the final two terms are dropped to make the branching even). As another example, image (b) in Figure 3.7 has 24 leaves, and $b = 8$. Therefore there are 3 internal nodes, each of which has 8 children. We hypothesize that the branching factor b has a negative correlation with execution time: as b increases, execution time should decrease. We use the *ratio* of the branch factor b over the sum of terms (feedback terms + original query terms) as the random variable used for testing. In order to control for as many variables as possible, a sample is generated by grouping the data points by method (either MAXSCORE or WAND), query id, and the number of feedback terms. We then calculate the correlation between the ratios

and times in the sample. Each correlation coefficient is then used as a data point in the analysis.

3.3.2 Results

Figure 3.8 shows a plot of the sample correlation coefficients. The majority of the points lie below -0.36 (the third quartile), and the median and mean values are -0.58 and -0.4853, respectively. From this plot we can infer that most queries benefit from increased exposure of the scoring nodes, however a small percentage of queries actually suffer from more opportunities to prune. The most likely explanation for the positive correlations is that these queries are “hard” queries, in that they rarely or never trigger pruning. In this situation, the overhead incurred when trying to prune simply slows execution down, and adding more checks that never trigger pruning only serve to exacerbate the problem.

3.4 Study 3: Ordering by Impact over Document Frequency

In the original MAXSCORE and WAND algorithms, the scorers are ordered according to the estimated lengths of their posting lists, the intuition being that if pruning occurs by the nodes processed first during a scoring pass, then moving the shortest posting lists to the front should minimize the number of documents considered. In a flat-tree model, particularly one with no weights, this logic makes sense. However our model has scoring nodes scaled by weights. Not only must we consider weights now, but as we rescale the weights of the nodes managed by the algorithm,

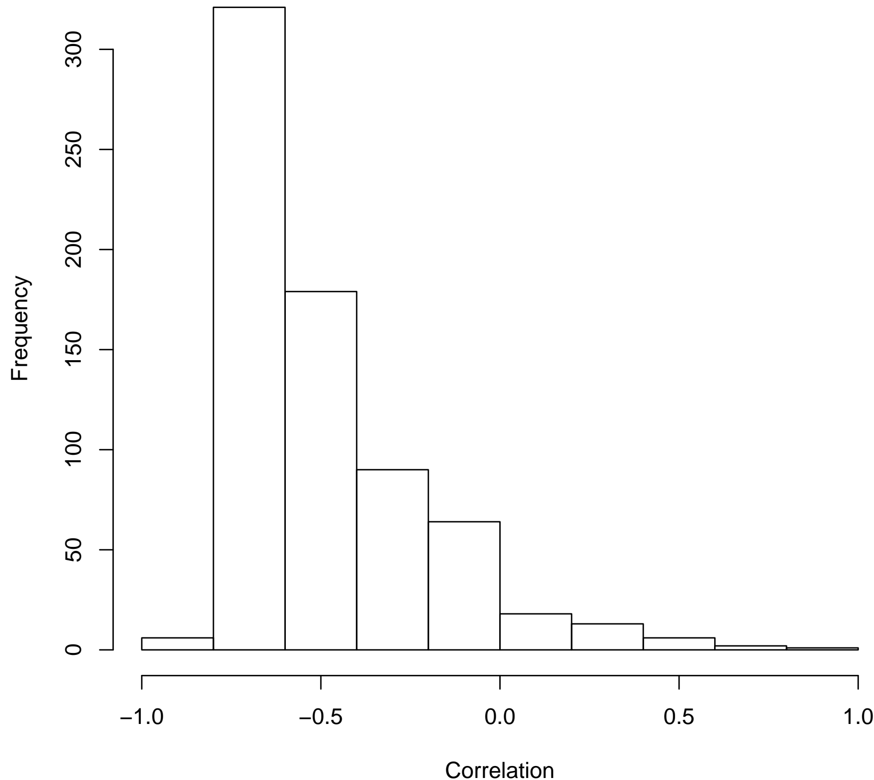

Figure 3.8. A plot of sample correlation coefficients between the *ratio* and *time* variables. Most queries show a significant negative correlation.

as we do in the MAX-* algorithms, we often have the situation that some subset of the nodes have weights orders of magnitude larger than the others.

Considering these new factors, we explore the possibility of ordering by weight to activate pruning sooner. We reason as follows: If the pruning decision is contingent on the heavily weighted nodes (i.e. we always prune when we hit the heavy-weight

node), then if we order by length of the posting lists, this weight imbalance is ignored. Ordering by weight pushes these nodes to the front of the scoring list, removing the wasted effort of scoring nodes that do not trigger pruning. We test this hypothesis on the Max-Flat algorithm, and call it Max-Flat-W. Results are shown in Table 3.4 for different values of λ, and for $\tau = 100$.

Method	Max-Flat	Max-Flat-W	% Chg
$\lambda = 0.2$	1876666.2	3430090.9	+82.3
$\lambda = 0.5$	1248424.9	1795394.2	+43.8
$\lambda = 0.8$	1053798.0	615244.3	-41.6
$\tau = 100$	39932114.8	37111540.1	-7.1

Table 3.4. Comparing list length and weight ordering for the Max-Flat algorithm.

Surprisingly, ordering by weight is only effective when $\lambda = 0.8$ or when $\tau = 100$. This indicates that the weight imbalance between the subquery components must be quite high for it to be effective. Worse performance of Max-Flat-W when $\lambda = 0.2$ supports this hypothesis. This weight brings the individual component weights closer to uniformity, reducing the importance of ordering the scoring list by weight.

3.5 Conclusion of Analysis

The experiments conducted all point to the same conclusion: blind application of optimization algorithms such as Maxscore or WAND, regardless of the query structure, can sacrifice signficant opportunity for further reducing the execution cost of the query. Additionally in some cases, the overhead incurred during optimization outweighs the saved work, resulting in a drop in amount of scoring work done but an increase in wall-clock time.

Second, we explored the effects of increasing exposure of the query to pruning via flattening. We found that increasing exposure of scoring nodes will most likely improve execution time. For a small number of queries, in particular queries that do not provide many opportunities to prune documents, the opposite effect occurs - increased exposure causes slower execution. We infer that this comes from the added overhead incurred when using a more complex scoring regime.

Thirdly, we investigated the effect of modifying the order in which the scorers are evaluated in the Maxscore algorithm. In the cases of extreme weight imbalance, ordering the scorers by weight instead of document frequency can result in substantial savings in execution time.

As a final point, the disparity in results between score-counting and timing is eye-opening. It is easy to make the assumption that instrumenting counts to function calls provides a reasonable surrogate to execution time; however the difference in results between Tables 3.1 and 3.2 plainly shows that no such assumption can be made until it is verified via empirical data.

CHAPTER 4

ALTERNATIVE SCORING REPRESENTATIONS

Chapter 3 performed an exploration of the effectiveness of the Maxscore and WAND processing algorithms with respect to changes in query depth. One general theme is already clear: the flatter we can make the query graph, the more effectively these algorithms can prune unneeded results. Clearly, if we have a mutable interpolated subquery, from an operational standpoint the query can be executed faster if we remove the subquery and move the subquery's children to the parent of the subquery. n some cases, such as a query expanded with a number of unigrams, performing this action is straightforward. However, sometimes flattening cannot be done by simply shuffling operators in the graph.

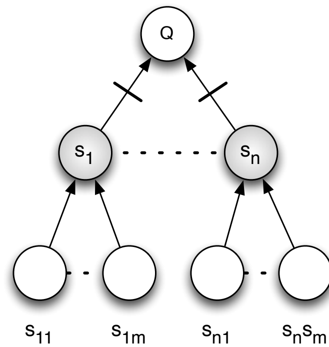

Figure 4.1. An example of a query graph that cannot be flattened.

Recall that both Maxscore and WAND view queries as flat, independent scoring functions that can be combined using a single composition function (e.g., addition). Given the current structure of Figure 4.1, the treatment of this query is:

$$Score(Q, D) = S_1 + \ldots + S_n \tag{4.1}$$

where the decision to prune lies in $n - 1$ places: before scoring s_i, for $i = 1 \ldots n$, respectively. These decision points are indicated in Figure **??** by the small slashes through the operator connections. Note that in this example, the calculation of each of these functions requires evaluating several scoring functions themselves – the result being that the real cost of evaluating the functions in Equation 4.1 is hidden from the pruning algorithm. One solution, similar to the approach taken by the CRM in (?, ?), is to make the costs of each of the composite functions explicit, and factor them into the pruning decision during evaluation. One immediate drawback of such an approach, however, is that each s_i may account for a significant portion of the final score, meaning that each decision to stop scoring (or skip scoring entirely, as in WAND), may be relatively "high-stakes". Since the CRM is not score-safe, then each component not scored (due to time constraints) may lead to large swings in score accuracy.

Suppose we are using Maxscore to prune as we score, and we have established at least k candidates. Therefore θ is a bounded number. We move to score document d. We first generate s_1, as described above. We make our update as before: $s_d = s_d + \text{Score}(d, q, I) - \text{UBnd}(q, I)$. We check that $s_d > \theta$ and find that the expression is false. The algorithm skips scoring the remaining terms, and moves to the next

candidate. If this decision could have been made after only determining s_{11}, we could have saved even more by not evaluating the rest of s_1. Under the current formulation, this is not possible for a such model.

Instead we may need to redefine how to express the query with a different set of operators that result in the same evaluation of a document, but with more independent scoring components. We call these re-expressions of the query *alternative scoring representations* (ASRs). The goal in constructing an ASR is to decompose the retrieval model operators into smaller parts, reformulating the complete scoring function into an equivalent one, composed of a higher number of simpler independent functions. Figure 4.2 provides a holistic view of the reformulation process.

Reformulating in this way carries several immediate advantages:

- We can directly apply the Maxscore and WAND algorithms to these new scoring functions without modification.
- Because the functions will be smaller, it is easier to compute upper-bound estimators for each function.
- We increase the granularity of the pruning mechanism during evaluation.
- We can isolate a "hotspot" in computation in the ASR function, instead of operations that may be spread over multiple operators. By having a single potential bottleneck, optimization efforts can focus on this single point.

Instead of trying to create a pruning algorithm equivalent to Maxscore or WAND, we look for a general procedure to reformulate a given scoring function into a math-

ematically equivalent form that is more responsive to pruning by providing a greater number of decision points during evaluation.

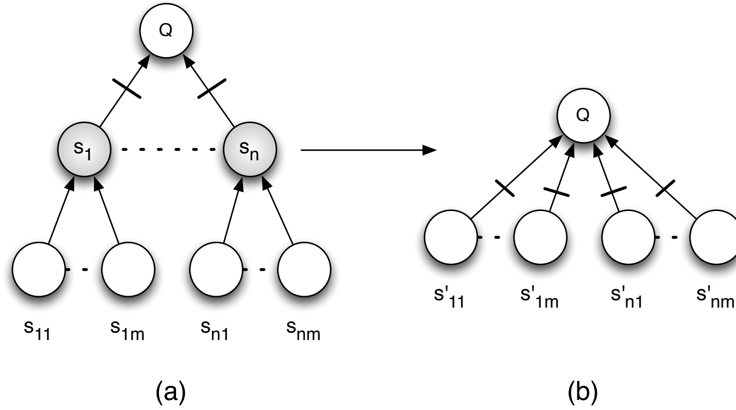

Figure 4.2. The generic idea of reformulating a query to allow for better pruning. Instead of attempting to prune after calculating every S_i (by aggregating over the sub-graph contributions), we rewrite the leaf scoring functions to allow pruning after each scorer calculation.

As an example, Fig 4.3 shows the execution rates of three different scoring algorithms for a chosen query[1] from one of our test collections. The three models shown are 1) exhaustive evaluation, where every candidate is fully scored (ex.), 2) Maxscore applied to the original formulation (ms-orig), and 3) Maxscore applied to the ASR of the given model (ms_F). As the graph shows, the pruning methods outperform exhaustive evaluation by spending, on average, less time on each of the candidate documents. The reformulated version (indicated in Fig 4.3 as "ms_F") performs even

[1] "fishing report 2006 jameson lake washington state"

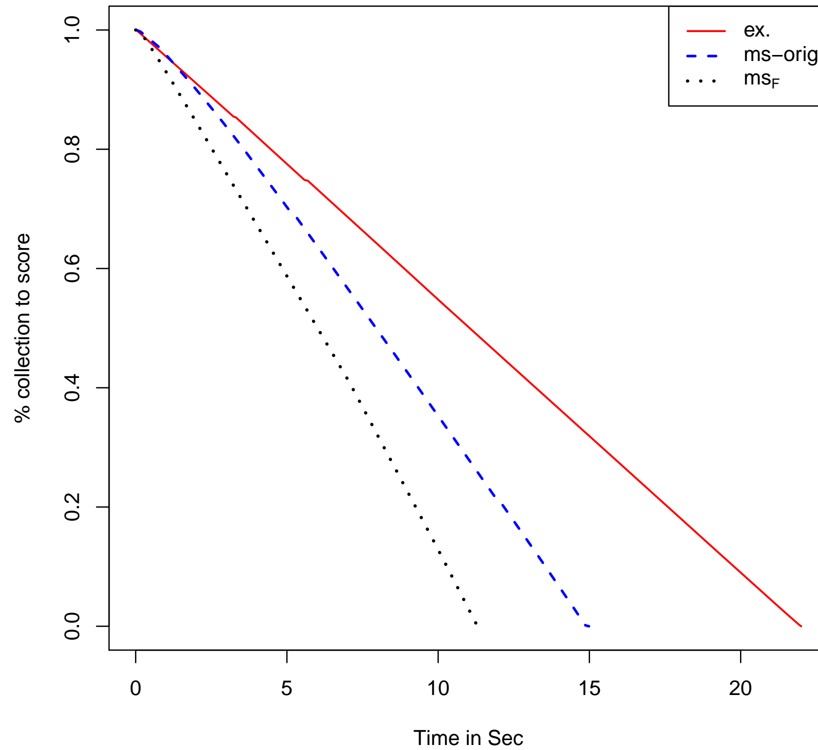

Figure 4.3. Different timings for Exhaustive, Maxscore (ms-orig) and Maxscore$_F$. The x-axis is time since the start of evaluation, and the y-axis is percent of the collection left to evaluate. The query evaluated is query #120 from the TREC Terabyte 2006 Efficiency Track.

slightly better than the original formulation by performing even fewer evaluations, thereby moving through the same number of candidates in less time.

In this chapter we show that ASRs can have the advantage of producing the same result as the original representation, but the alternative formulation responds better to the current approaches taken by dynamic pruning techniques, such as Maxscore and WAND.

After introducing the blueprint for producing ASRs, we then construct ASRs for two popular field-based retrieval models, PRMS and BM25F. We then present experimental results that show efficiency increases of up to 30% faster the original PRMS formulation providing strong evidence that the ASRs have the potential to afford the pruning algorithms much more opportunity to cut out unneeded work.

We continue with a brief discussion of the unexpected outcomes of the BM25F analysis, and conclude with current limitations of this approach.

4.1 Alternative Representation

We now develop the blueprint for developing ASRs. We begin by introducing the terminology used in later sections. We then proceed to describe ASRs both algebraically and operationally, and finally we derive ASRs for PRMS and BM25F.

4.1.1 Terminology

Let D denote the set of documents and $|D|$ the number of the documents in the collection. C denotes the multiset of term occurrences in the whole collection. $|C|$ is

then the number of tokens in the collection. For all formulae, we assume $d \in D, t \in C$. A query Q is composed of one or more terms. We assume $|Q| = n : Q = q_1 ... q_n$.

For a given retrieval model M, we desire to construct the ASR specific to M. We can think of M as a function that produces a score when given Q and $d \in D$, which we denote as S_{Qd}. We denote a document score using S, and a partial (i.e. leaf) score for d using lower case s. In all cases, when it is unambiguous we drop the d subscript for brevity: $s_{id} \rightarrow s_i$.

We assume that a candidate model for reformulation will contain scoring functions that are composites of lower-level scoring functions. For simplicity, we assume only one layer of composition, such as in Figure 4.1. We denote the lower level scorers as s_{ijd}, or simply s_{ij} when unambiguous.

As the examples in this chapter involve field-based models, we can extend the above notation to accommodate these models as well. For a given collection D, we assume there is a set of m specified fields: $F = \{f_1 \dots f_m\}$. For notational convenience, we define c_{ijd} as the number of times term i occurs in field j in document d, and analogously, s_{ijd} is the score for term i in field j in d (e.g. the score of `york` in the `publisher` field in document 37).

As in the previous chapters, we assume any scoring component can generate an upper bound (UBnd) and a lower bound (LBnd). We will make heavy use of the UBnd function in this chapter, therefore we use the following shorthand: for a given scorer s_i, we write the upper bound UBnd$(Q[i], I)$ as simply ϕ_i. We correspondingly use ξ_i for LBnd$(Q[i], I)$ when needed. Given the two-layer scoring construction discussed above, we also define ϕ_{ij} and ξ_{ij} to correspond to the upper

and lower bounds for s_{ij}, respectively. For consistency with the new symbols, we substitute the starting score quantities U_Σ and L_Σ (from Algorithms 2 and 5) with Φ and Ξ, respectively.

The following discussion will involve substitutions of the scorer bounds with actual scores. We denote the substitution of quantity ϕ_{ij} with s_{ij} in ϕ as $\phi[\phi_{ij}/s_{ij}]$ (and we may assume the analogous notation for ξ_{ji}). For generality, when we want to indicate a quantity that has some number of substitutions (including zero substitutions), we modify the symbol with a caret (e.g. $\phi \rightarrow \hat{\phi}$). When appropriate to quantify the number of substitutions made so far, we superscript the number of substitutions so far (e.g. $\hat{\phi}^3$ indicates 3 substitutions from the original ϕ). Again, we assume an analogous definition for ξ (i.e. $\hat{\xi}^3$).

For brevity in the following discussion, we only provide the ASR blueprint for operations using upper bounds, however the logic is nearly identical when we substitute lower bounds, except in the direction of the partial score. When starting from Φ, with each replacement the partial score non-increasingly approaches the true score S_{Qd}. When starting from the lower bounds (as in FindPivot), each replacement causes the partial score to non-decreasingly approach S_{Qd}.

4.1.2 Algebraic Description

Given a retrieval model M, we desire to construct $\digamma_M$, the ASR specific to M. $\digamma_M$ is the generic function for each s_{ij} that can provide the full update to the running estimate. In general we drop the subscript M from $\digamma^i_M$, as it is typically implied in the derivation. Using the substitution scheme defined above, we would like $\hat{\phi}^{nm} = S_d$, meaning after $n \times m$ substitutions, we arrive at S_d. We denote Φ as $\hat{\phi}^0$. Using these

equalities, we can express the transformation of Φ into S_d via a finite telescoping series:

$$\begin{aligned} S_d =& \left(\cdots \left(\hat{\phi}^0 + \left(\hat{\phi}^1 - \hat{\phi}^0 \right) \right) + \cdots + \left(\hat{\phi}^{nm} - \hat{\phi}^{(nm)-1} \right) \cdots \right) \\ =& \hat{\phi}^0 + \sum_{i=1}^{nm} \left(\hat{\phi}^i - \hat{\phi}^{i-1} \right) = \hat{\phi}^{nm} = S_d \end{aligned} \tag{4.2}$$

The summation form of the series (Equation 4.2) provides a clean analytical representation of S_d using bound replacement. We have $\hat{\phi}^0$ in hand (recall all documents start with this score), therefore we need to figure out $\left(\hat{\phi}^i - \hat{\phi}^{i-1} \right)$ for any $i > 0$. This quantity is the core of the ASR. For model M, for a given u and v s.t. $1 \leq u \leq n$ and $1 \leq v \leq m$, we define

$$\digamma^i_M = \left(\hat{\phi}^i_M - \hat{\phi}^{i-1}_M \right) = \hat{\phi}^{i-1}_M [\phi_{uv}/s_{uv}] - \hat{\phi}^{i-1}_M \tag{4.3}$$

which allows us to rewrite Equation 4.2 as

$$S_d = \hat{\phi}^0 + \sum_{i=1}^{nm} \digamma^i_M$$

The only difference between the two terms of $\digamma^i$, $\hat{\phi}^i$ and $\hat{\phi}^{i-1}$, is a single substitution. Our goal, for any given M, is to determine the effect of that substitution on $\hat{\phi}$. Also notice that u and v are not defined in terms of the previous substitutions, but just as adjustments to the current estimate $\hat{\phi}^{i-1}$. This means that we may order the

$\mathcal{F}^i$ functions in any way we like when scoring, and each replacement produces a decidable estimate. Put into a AI-style search context, each $\mathcal{F}^i$ acts a single step in a multi-step descent function, where the minimum (or maximum) extremum is the actual score. Similar to both Maxscore and Wand, if a document is fully scored, it is entered as a candidate for the final top k, and its score is accurate. Therefore this technique is both rank-safe and score-safe up to rank k.

4.1.3 Operational Description

We now show the change from an algorithmic perspective. We refer back to Maxscore, shown below (Algorithm 6). In the original examination of Maxscore, we made no claims about the efficiency of the SCORE function. Suppose now that for a given scorer $Q[i]$, the SCORE function is actually $\mathcal{O}(m)$, linear in the size of m, where m is based on some externally defined input (e.g. some number of automatically determined synonyms, or a set of fields, as we use later). Although it is not visible in Algorithm 6, the complexity of scoring a document is really $n * \mathcal{O}(m)$, with $n = |Q|$.

When constructed using the ASR of the model, $|Q|$ is now nm, and the SCORE function is constant ($\mathcal{O}(1)$), so although the full cost of scoring a document is the same ($nm * \mathcal{O}(1)$), we have an explicit representation of the cost with respect to the algorithm. The control mechanisms in Maxscore (and Wand) operate by determining how much of Q is valid. By increasing $|Q|$ from n to nm, the set of sentinels can be whittled down much more effectively, and in the case of Maxscore, more scorers can be skipped during evaluation.

Algorithm 6 Maxscore DAAT scoring algorithm.

```
MAXSCORE(Q, I, k)
 1  Q = INCDFSORT(Q, I)
 2  SN = Q
 3  U_Σ = Σ_{q∈Q} UBND(q, I)
 4  R = {}
 5  θ = −∞
 6  while SN has unfinished terms
 7       d = MINIMUMCANDIDATE(SN, I)
 8       s_d = U_Σ
 9       foreach q ∈ SN
10            s_t = SCORE(d, q, I)
11            s_d = s_d + s_t − UBND(q, I)
12       foreach q ∈ {Q − SN}
13            if s_d < θ
14                 abandon scoring of document d, and
15                      resume from step 6
16            s_t = SCORE(d, q, I)
17            s_d = s_d + s_t − UBND(q, I)
18       if s_d > θ
19            INSERT(R, ⟨docid = d, score = s_d⟩)
20            if |R| > k
21                 DEQUEUE(R)
22                 θ = R.head.score
23                 SN = SETSENTINELS(Q, U_Σ, θ)
24  R′ = REVERSE(R)
25  return R′
```

4.2 Field-Based Retrieval Models

There are many ways to model structure in documents for retrieval purposes. The Database (DB) community specializes in the case where documents are completely structured (i.e. every piece of data falls into a specific, well-defined field). Even considering semi-structured data, there has been research that bridges work between the DB and IR community, such as similarity joins (Cohen & Hirsh, 1998). However this approaches structured documents from the fully-structured side, and relaxes that constraint. Coming from the IR side of the space, we consider documents to have no structure, and therefore approach such data by augmenting structureless models by incorporating elements of structure.

Numerous researchers have spent years working on constructing usable digital libraries (Entlich et al., 1997; Fox & Sornil, 2003; Yi, Allan, & Croft, 2007), where the elements in the collection carry a considerable amount of structured metadata. The Initiative for the evaluation of XML (INEX) has engaged researchers in search over XML documents for several years, spurring research over documents with a hierarchical structuring of fields (Fuhr, Gövert, Kazai, & Lalmas, 2002). Addressing XML-type documents directly would represent a complete departure from the prior work shown in Chapter 3. Instead, we start with models that only consider first-order hierarchy (i.e. independent fields in the document). Towards this end, we focus on three recent probabilistic ranking retrieval models that have shown promise as natural extensions of several well-known IR retrieval models.

4.2.1 PRMS

The PRMS model was developed by Kim and Croft (Kim & Croft, 2009, 2010) in order to improve search over desktop-type collections. The PRMS model uses inferred field-mapping probabilities to weight the importance of each term/field pair in a given query. This is in contrast to the mixture of language-models (MLM) approach first proposed by Ogilvie and Callan (Ogilvie & Callan, 2003), where the component model weights are externally parameterized. For the purposes of this work, we may view PRMS and MLM as the same formulation; therefore the ASR derivation of PRMS applies to MLM without modification. Assuming m fields and n query terms, the score for a document d is the likelihood that d would generate the query Q:

$$P(Q|d) = \prod_{i=1}^{n} \sum_{j=1}^{m} P_{\mu}(j|q_i) P(q_i|j, d) \tag{4.4}$$

where P_{μ} is the "mapping probability" that field j is involved in the relevance estimation, given that q_i appeared. The probability is estimated as follows:

$$P_{\mu}(j|q_i) = \frac{P(q_i|j)P(j)}{\sum_{f_k \in F} P(q_i|f_k)}$$

We assume a uniform prior distribution for $P(j)$.

4.2.2 BM25F

The BM25F model was first developed by Robertson et al. as a response to many models at the time linearly combining scores. The authors showed that when a retrieval model uses a saturating term scoring function (such as BM25), scoring the fields independently results in the term appearing novel to each field. As the

first occurrence of a term provides more confidence than subsequent occurrences, scoring fields independently overweights the importance of a term. Instead, the authors combine at the term frequency level, then apply the scoring function over the combined frequencies. This leads to a much smoother rise in scoring as a function of term frequency across fields. We follow the BM25F formulation put forth by the same group of researchers for the TREC 2004 HARD Track (Zaragoza et al., 2004):

$$s_{ijd} = \frac{c_{ijd}}{(1 + B_j(\frac{l_{jd}}{l_j} - 1))} \tag{4.5}$$

$$s_{id} = \sum_{j=1}^{m} W_j s_{ijd} \tag{4.6}$$

$$\bar{s}_{id} = \frac{s_{id}}{K + s_{id}} idf_i \tag{4.7}$$

$$BM25F(Q, d) = \sum_{t \in q \cap d} \bar{s}_{id} \tag{4.8}$$

where l_j is the average length of field j across all documents, B_j is a field-specific tuning parameter, W_j is the weight for s_{ijd}, idf_i is the inverse document frequency of term i, and l_{jd} is the length of field j in document d. We now derive ASRs for PRMS and BM25F.

4.2.3 Rewriting PRMS

In order to find $\mathcal{F}_{PRMS}$, we first make sure we have, or can find, an admissible estimator for the model. $P(q_i|f_j, d)$ in Equation 4.4 is a language model estimate using either Dirichlet or Jelinek-Mercer smoothing. In the terminology introduced

in Section 4.1.1, $P(q_i|f_j, d)$ is s_{ijd}. Macdonald et al. derived max_{tf} as an admissible UBE for this scoring function (Macdonald et al., 2011), and we use that estimator for ϕ_{ij} here. The complementary lower bound value, ξ_{ij}, is generated by setting $c_{ijd} = 0$, and using the length of the longest document with respect to field j. We can now define the document-level potential of PRMS:

$$\phi = \prod_{i=1}^{n} \sum_{j=1}^{m} P_{\mu}(f_j|q_i)\phi_{ij} \tag{4.9}$$

Which is the value we start scoring a document from. In the following, we use w_{ij} to refer to $P_{\mu}(f_j|q_i)$, and let $\hat{\phi}_i = \sum_{j=1}^{m} w_{ij}\phi_{ij}$ to simplify the derivation. We start with the generic form of $\digamma_{PRMS}$:

$$\digamma_{PRMS} = \hat{\phi}^{i} - \hat{\phi}^{i-1} = \hat{\phi}^{i-1}[\phi_{uv}/s_{uvd}] - \hat{\phi}^{i-1} \tag{4.10}$$

We need to concretely define $\hat{\phi}^{i-1}[\phi_{uv}/s_{uvd}]$. Given the formulation in Equation 4.9, when $i \neq u$, $\hat{\phi}_i$ is a constant over the course of the substitution, as we are only replacing the value for term i in field j. Therefore, the components of term $i+1$ or term $i-1$, for instance, remain unchanged. So we only need to consider the substitution's effects on $\hat{\phi}_u$. The composite term score is a sum of term/field estimates, therefore the substitution involves subtracting $w_{uv}\phi_{uv}$ and adding $w_{uv}s_{uvd}$:

$$
\begin{aligned}
\hat{\phi}_u[\phi_{uv}/s_{uvd}] &= \left(\sum_{j=1}^{m} w_{uj}\phi_{uj}\right)[\phi_{uv}/s_{uvd}] \\
&= \left(\sum_{j=1}^{m} w_{uj}\phi_{uj}\right) - w_{uv}\phi_{uv} + w_{uv}s_{uvd} \\
&= \hat{\phi}_u + w_{uv}(s_{uvd} - \phi_{uv})
\end{aligned}
$$

We now define $\Phi = \left(\prod_{i=1}^{n} \hat{\phi}_i\right)$, the product of all term estimates, and by extension $\Phi_{-u} = \left(\prod_{i=1, i\neq u}^{n} \hat{\phi}_i\right)$, which is the product of all term estimates, excluding $\hat{\phi}_u$. Note that $\Phi = \hat{\phi}_{PRMS}$ (the total estimate, possibly with substitutions). This allows us to rewrite Equation 4.10 compactly:

$$
\begin{aligned}
F_{PRMS} &= \hat{\phi}^i - \hat{\phi}^{i-1} = \hat{\phi}^{i-1}[\phi_{uv}/s_{uvd}] - \hat{\phi}^{i-1} \\
&= \Phi_{-u}\left(\hat{\phi}_u + w_{uv}(s_{uvd} - \phi_{uv})\right) - \Phi \\
&= \Phi_{-u}\left(\hat{\phi}_u + w_{uv}(s_{uvd} - \phi_{uv})\right) - \Phi_{-u}\hat{\phi}_u \\
&= \Phi_{-u}\left(\hat{\phi}_u + w_{uv}(s_{uvd} - \phi_{uv}) - \hat{\phi}_u\right) \\
&= \Phi_{-u} w_{uv}\left(s_{uvd} - \phi_{uv}\right)
\end{aligned}
$$

The result is once again intuitive. Mulitplied out, the weighted potential ϕ_{uv} is removed from the total estimate, while the actual weighted contribution s_{uvd} is being added. The weight is the term/field weight w_{uv} multiplied by the remaining term

estimates. This quantity is fairly compact, but in implementation it can be difficult to correctly maintain Φ_{-u}. We can remedy this by involving $\hat{\phi}_u$ again:

$$\mathcal{F}_{PRMS} = \Phi_{-u} w_{uv} \left(s_{uvd} - \phi_{uv} \right) =$$
$$\left(\frac{\hat{\phi}_u}{\hat{\phi}_u} \right) \Phi_{-u} w_{uv} \left(s_{uvd} - \phi_{uv} \right) = \Phi \left(\frac{w_{uv}(s_{uvd} - \phi_{uv})}{\hat{\phi}_u} \right)$$

We can easily maintain cumulative scores for each term (i.e. $\hat{\phi}_i$), and we already have access to Φ, the total estimate. In this form, $\mathcal{F}_{PRMS}$ reduces to multiplying the current estimate by a small factor. As each replacement is a small negative number, the total probability slightly drops as we iterate through the different term/field pairs.

4.2.4 Rewriting BM25F

BM25F is an extension of the BM25 retrieval model. Instead of scoring fields independently, the term frequencies per field are first combined, then scored (S. Robertson et al., 2004). We begin by considering the set of formulae that constitute the BM25F scoring function:

$$s_{ijd} = \frac{c_{ijd}}{(1 + B_j(\frac{l_{jd}}{l_j} - 1))} \tag{4.11}$$

$$s_{id} = \sum_{j=1}^{m} W_j s_{ijd} \tag{4.12}$$

$$\bar{s}_{id} = \frac{s_{id}}{K + s_{id}} idf_i \tag{4.13}$$

$$BM25F(Q,d) = \sum_{t \in q \cap d} \bar{s}_{id} \tag{4.14}$$

where l_j is the average length of field j across all documents, B_j is a field-specific tuning parameter (analogous to the B parameter in BM25), W_j is the field weight, idf_i is the inverse document frequency of term i, and l_{jd} is the length of field j in document d. As before, we first find an admissible estimate for the term/field scoring function. In this model, the term/field scoring function corresponds to Equation 4.11.

Finding Bounds for BM25F's Nodes

We would like to create a reasonable admissable estimate we can make for any term/field pair. We use the same technique as Macdonald et al. and view the problem as a relaxed constrained maximization problem. Let $x = c_{ijd}$, and let $y = l_{fd}$:

$$
\begin{aligned}
s_{ijd} =& \frac{c_{ijd}}{(1 + B_j(\frac{l_{jd}}{l_j} - 1))} & = \frac{x}{(1 + B_j(\frac{y}{l_j} - 1))} \\
=& \frac{x}{(1 + B_j(\frac{y}{l_j} - 1))} & = \frac{x}{1 + \frac{B_j y}{l_j} - B_j} \\
=& \frac{x}{\frac{B_j y}{l_j} + (1 - B_j)} & = \frac{x}{\alpha y + \beta}
\end{aligned}
$$

where $\alpha = B_j / l_j$ and $\beta = (1 - B_j)$. We assume both x and y have lower and upper bounds (e.g. x_{max} is the maximum x for any document). We would like to maximize s_{ijd} subject to

- $x \leq y$
- $x_{min} \leq x \leq x_{max}$
- $y_{min} \leq y \leq y_{max}$
- $0 < x_{min} < x_{max}$
- $0 < y_{min} < x_{max} < y_{max}$

This function monotonically increases w.r.t. x and monotonically decreases w.r.t. y, very much like the classic functions studied before (Macdonald et al., 2011). Given our constraints, this function is maximized when $x = y$, since a constraint is that

$x \leq y$, and in the case of $x < y$, we can find a larger value by increasing x towards y. We substitute x for y, and take the derivative w.r.t. to x to get

$$\frac{\partial s_{ijd}}{\partial x} = \frac{\beta}{(\alpha x + \beta)^2} \tag{4.15}$$

which is a positive-valued function $\forall x \geq 0$. Therefore we can follow the gradient produced in Equation 4.15 to increase the value of x until we reach its maximum allowable value as defined by the given constraints. This value is $\text{argmax}_{d \in D}\, c_{ijd}$, which we label as $\bar{x}$. Based on this derivation, we now have an admissible estimator for Equation 4.11. Therefore, for a given term, the average length of field j across documents, and parameter B_j for field j, we define our estimator as:

$$\phi_{ij} = \frac{\bar{x}}{(1 + B_j\left(\frac{\bar{x}}{l_j} - 1\right)}$$

Given the form of ϕ_{ij}, and indeed the BM25 term scoring function general, it is not difficult to see that $\xi_{ij} = 0$ (simply set $c_{ij} = 0$). We now proceed to derive $\digamma_{BM25F}$.

Deriving the ASR for BM25F

Substituting term/field potentials for scores in Eqs. 4.11-4.14, we define potentials for BM25F as follows:

$$\phi_i = \sum_{j=1}^{m} W_j \phi_{ij}$$
$$\bar{\phi}_i = \frac{\phi_i}{K + \phi_i} idf_i$$
$$\phi = \sum_{t \in q \cap d} \bar{\phi}_i$$

We start with the general version of $\digamma_{BM25F}$:

$$\digamma_{BM25F} = \hat{\phi}^i - \hat{\phi}^{i-1} = \hat{\phi}^{i-1}[\phi_{uv}/s_{uvd}] - \hat{\phi}^{i-1} \tag{4.16}$$

As before, when making a single substitution, all $\hat{\psi}_i$ where $i \neq u$ are held constant during the substitution, and can be ignored. This leaves us with:

$$\begin{aligned} \digamma_{BM25F} &= \hat{\psi}_u^{i-1}[\phi_{uv}/s_{uvd}] - \hat{\psi}_u^{i-1} \\ = \left(\frac{\phi_i[\phi_{uv}/s_{uvd}]}{K + \phi_i[\phi_{uv}/s_{uvd}]} idf_u\right) - \left(\frac{\phi_i}{K + \phi_i} idf_i\right) &= \\ \left(\frac{\phi_i[\phi_{uv}/s_{uvd}]}{K + \phi_i[\phi_{uv}/s_{uvd}]} - \frac{\phi_i}{K + \phi_i}\right) idf_u \end{aligned} \tag{4.17}$$

If we consider the semantics of $\hat{\phi}_u[\phi_{uv}/s_{uvd}]$ w.r.t. BM25F, we find that the substitution has a similar effect as in the PRMS case:

$$\hat{\phi}_u[\phi_{uv}/s_{uvd}] = \hat{\phi}_u + W_v(s_{uvd} - \phi_{uv})$$

As both formulae use sums over the fields to generate the term values, this similarity is expected. Substituting into Equation 4.17, we have:

$$\mathcal{F}_{BM25F} = idf_u \left(\frac{\hat{\phi}_u + \xi_{uv}}{K + \hat{\phi}_u + \xi_{uv}} - \frac{\hat{\phi}_u}{K + \hat{\phi}_u} \right)$$

where $\xi_{uv} = W_v(s_{uvd} - \phi_{uv})$. As in the case of $\mathcal{F}_{PRMS}$, we must maintain $\hat{\phi}_i$ for all $q_i \in Q$. However using this value it is easy to compute the value of $\mathcal{F}_{BM25F}$.

4.3 Experiments

Now that we have defined ASRs for PRMS and BM25F, we carry out experiments to determine the difference in performance between the original models and their ASRs. Given a collection C and set of queries Q, we would like to compare the execution times of exhaustive evaluation (Exhaustive), where all candidates are fully scored, Maxscore using the original retrieval function, and Maxscore using the ASR (Maxscore$_{\mathcal{F}}$) for Q executed over C. We run identical experiments for Wand and Wand$_{\mathcal{F}}$ as well.

Data

We conduct experiments over two different data sets: 1) the first 250 queries from the Efficiency task query set from the TREC Terabyte 2006 track, over the GOV2 document collection (TB06), and 2) a download of the metadata records from the OpenLibrary (OL), along with a sample from 10000 queries from the OpenLibrary query logs. To reduce noise in the queries selected from the OL log, we simplify by only using queries that are length 7 or shorter, and we disallow queries that contain

repeat terms, leaving the set with slightly over 9000 queries. We received the queries in alphabetical order, therefore to reduce bias on a particular part of the vocabulary, we randomized the query list using the `shuffle` method of the Python 2.7.3 `random` module, initialized with a seed of 0. We then select our experimental 250 queries from the head of this randomized list.

As we are only retrieving unigrams in these models, we stem the queries using the Porter stemmer, and we exclude stopwords from the INQUERY 418 stopword set. Table 4.1 contains some statistics about each of the collections considered. TB06 documents have a `title` field, the body content stored as the `body` field, and the anchor text indexed as an additional field `a`. TB06 provides a relatively standard case of web pages, with only a few fields of interest, and one of the fields significantly outweighing the others in terms of content density (i.e. the `body` field is significantly bigger than the `title` or `anchor` fields).

The OL records provide a contrastive dataset - there are 22 fields, none of which contain significantly more content than the rest. The records are community-built, therefore not all fields are present in all documents. The simple statistics in Table 4.1 indicate the differing structure in OL versus TB06. The average number of tokens per field in OL is much lower than in TB06. Despite the larger number of fields, the standard deviation of the number of terms in each field is also lower. Kim et al. perform an in-depth analysis of this dataset[2]

[2]This dataset can be downloaded at: `http://ciir.cs.umass.edu/~hfeild/downloads.html#openLibrary2012`

Test Set	# docs	terms	fields	(Tokens/Field) / σ
Scale	M	M	-	M
TB06	25.2	22,333	3	5,304 / 5,870
OL	39.4	1,418	22	64.4 / 85.2

Table 4.1. Statistics on the collections used in experiments. 'M' indicates a scale of millions. The last column shows the average number of tokens per field for that collection. The second value in that column is the standard deviation of the distribution of tokens per field.

Software and Hardware

We conduct experiments using a modified version of the Galago v3.3 retrieval system. The index structure is a positional index, with term-interleaved document, count, and position blocks. The document ids and positions for a given document are d-gapped, with all integers and longs compressed using vbyte encoding. Skips are implemented using a separate two-level skiplist structure. High-level skips record the absolute byte position every 10,000 postings, with relative skips being recorded for each block every 500 postings. If a position block contains more than 2 entries, it is prepended with the list length in bytes, to allow for skipping the block without decompressing it.

All runs were conducted on one core of a 2.66 GHz Intel Core 2 Quad-Core Q8400 with 8GB RAM. The index resides on a local disk (1TB 7200 rpm Seagate Barracuda with 32MB cache), and the operating system used is Linux Ubuntu 12.04.

Measurement

In order to minimize the effects of disk latency, for each query we perform a warmup run of the query to load as much data into memory as possible, and then we

immediately run the query again, and use the second time produced as the recorded time. The primary measurement is the *average latency* of a query (time to process that query). All times were measured in milliseconds.

To compare two runs (say the original PRMS formulation and F_{PRMS}), we report the average drop in query latency as a ratio of the original query latency. Let t_O and t_M be times produced by the original (O) and modified (M) runs, respectively. We calculate the ratio of improvement as t_M/t_O. Therefore a value of 0.9 indicates that run M required only 90% of the time taken by run O. A value > 1 indicates that run M ran longer than run O. Therefore, a lower value is better. We also report the number of queries improved or worsened compared to the original formulation.

We determine statistical significance via a two-sample permutation test as described by Efron and Tibshirani (Efron & Tibshirani, 1993). All results are verified to be statistically significant for $p < 0.01$ unless otherwise noted. We omit retrieval effectiveness results, as the pruning algorithms are by construction safe-to-rank-k. Therefore, the top k documents for a given query are receive the exact same score as the original models.

4.4 Results

Exhaustive vs. Standard Pruning Methods

Tables 4.2 and 4.3 show a breakdown of query performance by scoring algorithm and by query length for the TB06 and OL collections, respectively. The straightforward applications of Maxscore and Wand to the field-based retrieval model perform as expected: other than a query length of one, the pruning algorithms reduce com-

putational cost by a large margin, indicating that 1) we have a simple criteria for when to use Exhaustive scoring (i.e. on one-word queries), and 2) we now have strong empirical support for using the pruning algorithms for field-based models.

Standard vs $\mathcal{F}$-function Pruning

In comparing the original functions to $\mathcal{F}$-functions, we see two different behaviors emerge, evidently based on the scoring function used. Under PRMS, the $\mathcal{F}$-functions provide a consistent additional improvement even over the original pruning algorithm. However when using BM25F, the $\mathcal{F}$-functions perform slightly better when used in conjunction with Wand on the OL test set, but perform slightly worse on average in the other configurations. This lopsided behavior prompted us to perform several supplemental experiments and a brief code review to reduce the odds that a bug caused this behavior, but all the outcomes were consistent with the reported results.

To provide an alternate view between the original formulations and the $\mathcal{F}$-functions, we report the number of improved and worsened queries for each configuration in Table 4.4. Here we can see that the number of better or worse queries highly correlates with the averages shown in Tables 4.2 and 4.3. In both the TB06 and OL test sets, the $\mathcal{F}$-function improves the majority of queries for that set. Looking at BM25F, the $\mathcal{F}$-function tends to slightly hurt more often than it helps.

Performance Across Test Sets

Recall that the OL test set provides a different data profile than the TB06 set. Viewing the results in Tables 4.2 and 4.3, we can note that all pruning algorithms

TB06	BM25F						
Query length	1	2	3	4	5	6	All
Number of queries	5	56	82	63	31	13	250
Exhaustive	0.27	1.55	4.80	7.46	11.36	16.30	6.06
Maxscore	*0.94*	0.51	0.35	0.28	0.26	0.24	0.30
Maxscore$_F$	*0.90*	0.51	0.37	0.30	0.29	0.27	0.33
Wand	0.96	0.51	0.36	0.28	0.27	0.24	0.31
Wand$_F$	0.92	0.54	0.39	0.30	0.28	0.25	0.33

TB06	PRMS						
Query length	1	2	3	4	5	6	All
Number of queries	5	56	82	63	31	13	250
Exhaustive	0.22	1.14	3.95	6.70	10.68	16.02	5.40
Maxscore	*1.02*	0.66	0.50	0.39	0.35	0.41	0.42
Maxscore$_F$	*0.87*	0.46	0.37	0.32	0.30	0.32	0.34
Wand	*1.05*	0.66	0.51	0.32	0.27	0.26	0.36
Wand$_F$	0.88	0.53	0.43	0.28	0.24	0.22	0.31

Table 4.2. Relative scoring algorithm performance over the Terabyte06 collection, broken down by query length. Exhaustive times are reported in seconds, while other times are reported as a ratio of the Exhaustive time. All relative times are statistically significantly different from the baseline time, unless noted by italics.

have better relative efficiency over the OL set than the TB06 set, indicating that a more evenly spread, larger number of fields in the document collection provides a more favorable environment for pruning algorithms. The relationships between the original and ASR versions appears unchanged between the two data sets, indicating that the ASRs are robust to changes in the number of fields. The results also weakly suggest that shifts in the content density distribution (i.e. very lopsided like in TB06 vs. evenly spread in OL) do not affect performance of the pruning algorithms, however we believe this claim requires further future investigation.

OL	BM25F						
Query length	1	2	3	4	5	6	All
Number of queries	17	77	73	41	30	12	250
Exhaustive	0.05	1.15	5.33	34.16	49.26	117.18	19.05
Maxscore	0.97	0.87	0.29	0.08	0.07	0.05	0.10
Maxscore$_F$	0.98	0.84	0.31	0.10	0.09	0.09	0.12
Wand	0.98	0.84	0.25	0.07	0.06	0.04	0.09
Wand$_F$	*0.98*	0.77	0.25	0.07	0.05	0.03	0.08
OL	PRMS						
Query length	1	2	3	4	5	6	All
Number of queries	16	77	73	41	30	12	249
Exhaustive	0.01	0.55	3.39	24.12	31.63	83.93	12.99
Maxscore	0.87	0.79	0.21	0.07	0.06	0.04	0.08
Maxscore$_F$	0.84	0.64	0.18	0.06	0.04	0.02	0.06
Wand	0.86	0.80	0.19	0.05	0.04	0.02	0.06
Wand$_F$	0.84	0.40	0.09	0.02	0.02	0.01	0.03

Table 4.3. Relative scoring algorithm performance over the OpenLib collection, broken down by query length. Exhaustive times are reported in seconds, while other times are reported as a ratio of the Exhaustive time. All relative times are statistically significantly different from the baseline time, unless noted by italics.

Coll	Model	Method	# gain/loss	avg +	avg -
TB06	BM25F	Maxscore	47/199	0.967	1.091
		Wand	75/169	0.950	1.150
	PRMS	Maxscore	220/27	0.811	1.127
		Wand	155/90	0.787	1.428
OL	BM25F	Maxscore	79/162	0.934	1.161
		Wand	154/86	0.867	1.271
	PRMS	Maxscore	132/97	0.761	1.176
		Wand	216/15	0.594	1.131

Table 4.4. A breakdown of the number of improved (win) and worsened (loss) queries, by collection, scoring model, and pruning algorithm.

4.4.1 Bounds Sensitivity

As the results in Table 4.4 show, sometimes the ASRs provide a significant boost in efficiency, and in other cases, they seem to have no effect, or are more likely to slightly degrade efficiency. In an effort to better understand this behavior, we perform a simple experiment to examine whether the bounds estimation error has any impact on the apparent contrast in behavior.

For both PRMS and BM25F, we derived admissible bounds for both the lower and upper bounds of the scoring functions. While these are valid bounds, they are not necessarily tight bounds. Determining the error of the bounds is a simple matter - for a given scoring component, iterate over its entire posting list, and record the maximum and minimum[3] values encountered. We then compare the actual values with the ones estimated. In the following discussion we call these values the *actual* and *estimated* values, respectively. Figure 4.4 shows the $\log_2$ distributions of the ratio of the estimate over the actual value for the BM25F original and ASR formulations. Mathematically, this value is:

$$\log_2 \left(\frac{estimated}{actual} \right)$$

Therefore, the closer to 0 the value is, the closer the estimate is to the actual value. Figure 4.4 shows that the majority of the estimated bounds for the original formulation were very close to the actual value (less than twice as much as the actual).

[3]PRMS is a smoothed language model, meaning a term/field value should be above zero. We therefore include the minimum value when analyzing the bounds.

However for the ASR formulation, virtually all of the scoring components had estimates that were at least twice as much as the actual upper bound. The disparity between the PRMS formulations follows an opposite trend, although less extreme (Figure 4.5). In that case many of the $\mathcal{F}$ scorers are extremely, accurate, while many original scorers are closer to four times as much or more. Since both Maxscore and Wand rely on the UBE to determine when to prune, there indeed may be significant waste of effort due to overestimation.

The next step in our analysis involves re-executing the queries, but artificially setting the upper and lower bounds of the scoring components to the precomputed values before scoring begins. These runs are shown in Table 4.5. Runs in this table are directly compared to their counterparts that used the estimated values. Therefore a value of 0.5 indicates that the actual value run executed in 50% of the time of the estimated value run.

Surprisingly, the BM25F seems largely unaffected by using the actual bounds. The reported values indicate slightly worse execution times, but none of the results turn out to be statistically significant. The actual value PRMS runs exhibit a marked improvement over the estimated value runs, across all configurations. Although we learned that accurate bounds estimation is important for PRMS, is appears to be inconsequential for BM25F, and it provided no further illumination of the polarized behavior of the ASRs between PRMS and BM25F.

TB06	BM25F						
Query length	1	2	3	4	5	6	All
Number of queries	5	56	82	63	31	13	250
Maxscore	*0.99*	*1.01*	*1.04*	*1.07*	*1.04*	*1.07*	*1.05*
Maxscore_F	*1.04*	*1.03*	*1.04*	*1.08*	*1.05*	*1.08*	*1.06*
Wand	*1.00*	*1.00*	*1.01*	*1.02*	*1.01*	*1.02*	*1.01*
Wand_F	*1.00*	*1.00*	*1.01*	*1.02*	*1.01*	*1.02*	*1.02*

TB06	PRMS						
Query length	1	2	3	4	5	6	All
Number of queries	5	56	82	63	31	13	250
Maxscore	*0.99*	0.41	0.66	0.58	0.56	0.49	0.57
Maxscore_F	0.94	0.41	0.65	0.50	0.55	0.45	0.54
Wand	*1.01*	0.39	0.53	0.54	0.48	0.37	0.49
Wand_F	*1.00*	0.50	0.61	0.43	0.45	0.32	0.49

Table 4.5. Relative improvement of the actual value runs vs. the estimated value runs. Values are calculated as `actual / estimated`, therefore the lower the value, the greater the impact tight bounds has on the configuration.

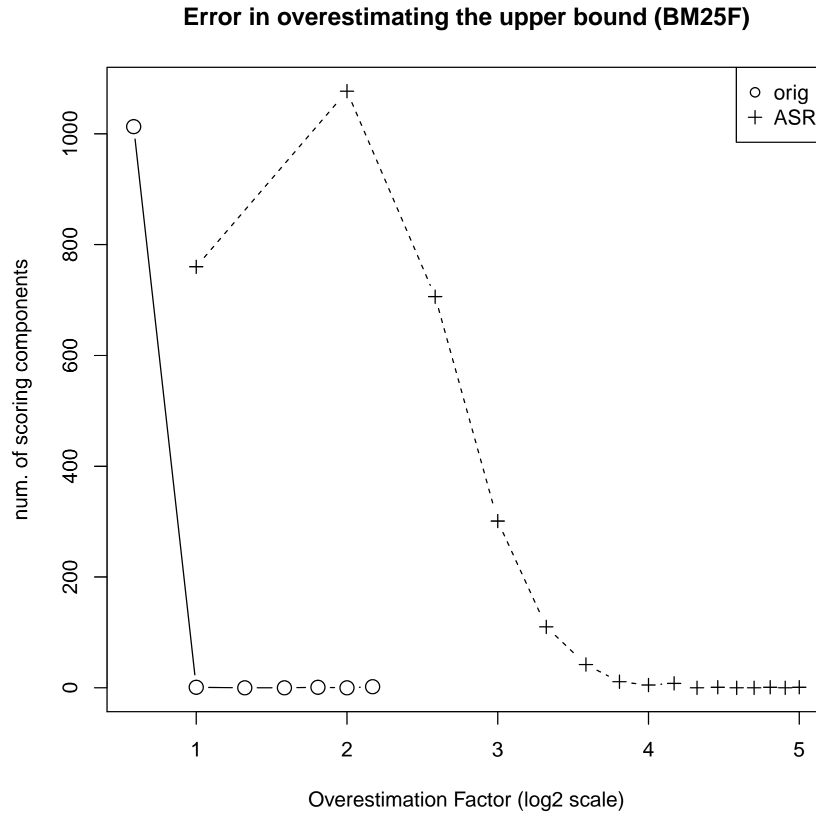

Figure 4.4. The error in the UBE overestimating the actual upper bound of the scorer. The graph above is of BM25F (both original and ASR formulation), over the first 200 queries of the Terabyte 2006 Efficiency track, using the GOV2 collection.

4.5 Current Limitations of ASRs

A clear limitation to ASRs is whether or not the original retrieval model can be algebraically massaged to fit into the correct form. Some functions simply cannot be rewritten, and they fall outside the scope of this optimization. Currently we are not aware of a simple litmus test to determine if a particular retrieval model has an

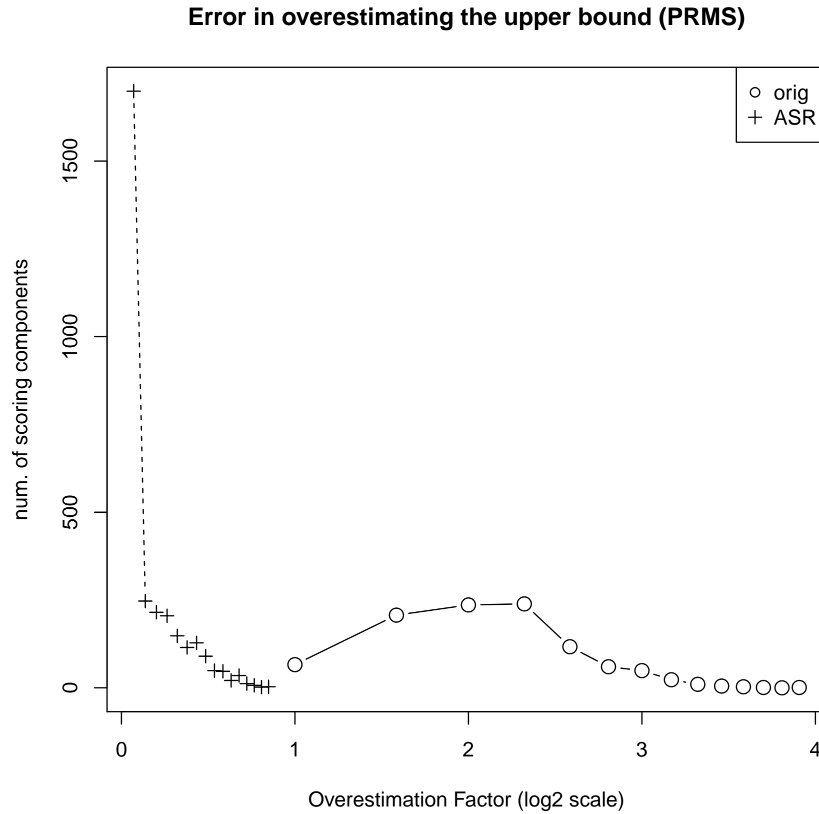

Figure 4.5. The error in the UBE overestimating the actual upper bound of the scorer. The graph above is of PRMS (both original and ASR formulation), over the first 200 queries of the Terabyte 2006 Efficiency track, using the GOV2 collection.

ASR or not. Exploration into whether this problem is easily decidable may be an interesting avenue for future research.

Until such a test exists, to construct an ASR, we will have to try to derive manually. This process may also prove to be a limiting factor, as it limits the chance that an implementor will spend the energy necessary to derive the function.

CHAPTER 5

STRATEGIC EVALUATION OF MULTI-TERM SCORE COMPONENTS

So far we have focused on increasing pruning opportunities by either restructuring or reformulating queries. We now turn our attention to a different kind of complexity in modern retrieval models: conjunction operators. Unlike an operator over a single index key (term), a conjunction operator relies on two or more keys in order to generate a value, and involves some calculation between the multiple keys that either must be precomputed and stored, or must be generated at query time. We define a *stored* component to be one that is explicitly contained in the index, and a *synthetic* component to be one that is constructed at query time using information from the index, or by processing some portion of the document collection. For example, in a unigram index a single word such as `bicycle` is a stored term. On the other hand, the phrase `"racing bicycle"` is a synthetic term – while it is not stored in the index in its own right, it can nevertheless be processed as part of a query by (assuming that term positions are indexed) constructing the intersection of the `racing` and `bicycle` posting lists. It would also be possible to include information about the occurrences of some or all of the bigrams occurring in the source text in the index, in which case `"racing bicycle"` might be a stored term. But in this latter case the index

would need to be significantly larger, and the execution-time savings would need to be carefully weighed against increased storage costs.

To mitigate this constraint, we propose using *staged evaluation* to minimize the overhead involved in evaluating conjunction operators at query time. Instead of trying to partially evaluate the synthetic operators, we instead aim to evaluate only when we are confident their contribution will matter. The work presented here is done in collaboration with Alistair Moffat, from the University of Melbourne, Australia.

The idea of evaluating a query in a staged fashion is not new. One prior approach would break the query apart to evaluate it in a "waterfall" fashion at different locations (Moffat, Webber, Zobel, & Baeza-Yates, 2007b). This approach tends to improve throughput of a query stream, but the authors had difficulty in reducing query latency (note that a single query would *have* to visit multiple processors to be fully evaluated). The purpose of this research was not to reduce the total amount of work done during evaluation, but to provide an alternative scoring regime in a distributed setting. Latency issues aside, this approach could be easily adapted to support partial evaluation by returning the partially processed query, however this would breach any guarantees regarding score correctness.

Another line of research uses separate stages to evaluate parts of a query, using learned costs for each stage to determine when scoring must cease in order to meet a time constraint (Arnt, Zilberstein, Allan, & Mouaddib, 2004; Wang et al., 2011). This results in a tradeoff between the accuracy of the resulting score and speed, with speed being the priority. Therefore, the question this approach looks to answer is: *Given a time constraint, how much of the scoring function can we evaluate?*. This

may make sense at first, as many SLA (service-level agreement) contracts are often phrased with time being a hard constraint. However the loss of guarantees with respect to the scoring functions necessarily limits our confidence in two important cases: 1) domain transfer, where knowledge gained about retrieval models and certain features is less certain as we stray further and further from the actual performance of the original model, and 2) use of search as a service to other systems. Using the philosophy described above, we can return a response quickly, but the confidence attached to that response is not only lower, in many cases it may be unknown – we do not know how far from the original ranking we actually are.

Another way to view the issue is to ask: *How long does it take to achieve a certain level of accuracy?* In this respect, fidelity to the original scoring function is the primary constraint, with time being the free variable we are trying to minimize. Based on this philosophy, we propose a staged evaluation scheme that is grounded in the conjunction constructs described in Section 2.4.3. By bounding the type of components we consider and examining their effects on query score, we may now consider the three distinct types of scoring fidelity, up to a rank k:

Set-Safety @ *k* The results contain the correct top k documents, however their order within the top set is not guaranteed.

Rank-Safety @ *k* The results contain the correct top k documents, and they are in the correct order. Their scores are not guaranteed to be accurate. This is a stronger version of the set-safety property.

Score-Safety @ k The top k documents are all scored correctly, therefore are in the right order. Note that as defined this does *not* imply set-safety @ k. That is, a document may not be scored at all, and therefore not placed in the top k, although it should appear.

The first step towards staged evaluation involves a low-cost initial pass to assemble a candidate set of results, along with some form of estimate of the remaining components to be calculated. From here, we can investigate multiple options for completing the evaluation, based on the requirements determined by the query issuer. Set-safe evaluation would most likely require the least amount of remaining evaluation, followed by rank-safe evaluation, score-safe evaluation, and finally both rank- and score-safe evaluation (i.e. all required documents in the returned set, all correctly scored). We now present the algorithmic details of staged evaluation over conjunction operators.

5.1 Deferred Query Components

Instead of fully instantiating all query components at the beginning of scoring, we propose a process in which score contributions are calculated for stored terms, and score contribution *intervals* are amassed for the synthetic terms in the query. The procedure – denoted as DeferredScore – is shown in Algorithm 7. Two sets of query components are maintained: ST is the set of stored terms, for which exact score computations can be performed; and, as in MaxScore, set SN is a set of sentinel terms that decreases in size as the threshold t for consideration as a candidate answer rises.

Algorithm 7 Scoring based on intervals.

```
DeferredScore(Q, I, k, R)
 1  SN = ST = {q | q ∈ Q and q ∈ I}
 2  U_Σ = Σ_{q∈Q} UBnd(q, I)
 3  R = R_k = {}
 4  t = −∞
 5  while SN has unfinished terms
 6      d = MinimumCandidate(SN, I)
 7      s_d = U_Σ
 8      min_d = max_d = 0
 9      foreach q ∈ Q
10          if (s_d + max_d) < t
11              abandon scoring of document d, and
12                  resume from step 5
13          if q ∈ ST
14              s_d = s_d − UBnd(q, I) + Score(d, q, I)
15          else
16              s_d = s_d − UBnd(q, I)
17              min_d = min_d + EstMin(d, q, I)
18              max_d = max_d + EstMax(d, q, I)
19      if s_d + min_d > t
20          Insert(R_k, ⟨d, s_d, min_d, max_d⟩, s_d + min_d)
21          if |R_k| > k
22              Delete(R_k, k + 1)
23              t = R_k[k].score + R_k[k].min
24              SN = SetSentinels(ST, U_Σ, t)
25              while R[|R|].score + R[|R|].max < t
26                  Delete(R, |R|)
27      if s_d + max_d > t
28          Insert(R, ⟨d, s_d, min_d, max_d⟩, s_d + max_d)
29  R = FinalizeScores(Q, I, k, R)
30  return R
```

In the new implementation the set R holds tuples of type

$$\langle d, S_d, min_d, max_d \rangle,$$

accessed by the attributes *id*, *score*, *min*, and *max*, respectively. The primary heap R now holds a set of k or more items, any of which might, based on the information processed to date, be amongst the k highest scoring documents when all components of the evaluation are complete. A "top heap", shown as R_k, is added duplicating k of the elements in R.

As well as being different in size, the two heaps are also arranged differently – the ordering of R_k is based on $s_d + min_d$, so that R_k contains the k documents for which it is "quite probable" that they will be among the eventual top k; whereas the ordering of R is based on $s_d + max_d$, and it contains any other documents processed so far for which it is "not yet impossible" that they might challenge for a spot in the top k, once all synthetic terms are fully resolved.

To reinforce the intuition underlying DeferredScore, consider the 10 example documents shown in Table 5.1. Supposed we have a query with two query components: q_1, a stored term, and q_2, a synthetic term. The component scores for these two terms are shown in the columns headed q_1 and q_2. In our approach, however, we do not know the actual value of q_2 because the synthetic posting list has not been materialized. As a result, we do not have the collection-level statistics needed to estimate q_2 (for example, document frequency). Instead, we use estimators – called EstMin and EstMax in Algorithm 7 – that use statistics at hand to provide bounds on the value. Column q_2' shows possible bounds; remember that q_2 itself is actually

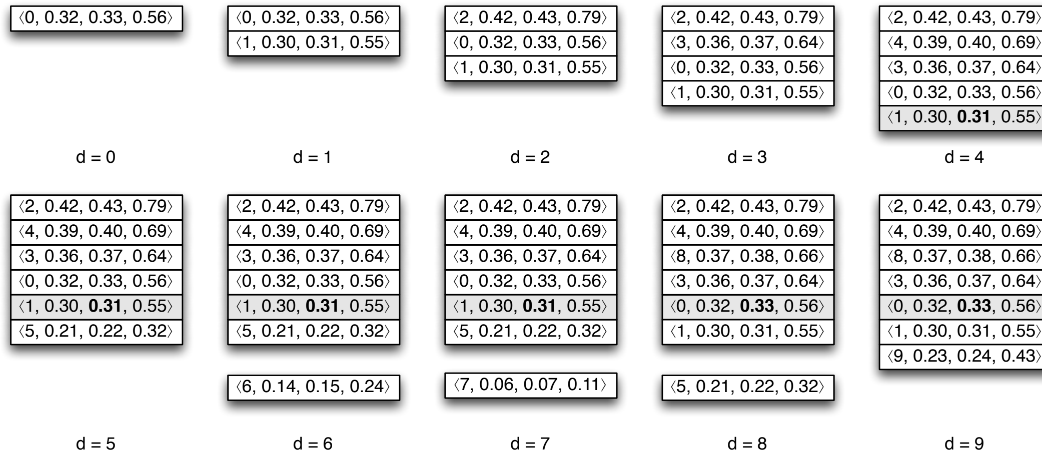

Figure 5.1. Contents of R_k after evaluating each document from Table 5.1. The grey entry indicates the candidate at rank $k = 5$, which is used for trimming the list when possible. The top k elements are also stored in a second heap R_k of size k, ordered by the third component shown, min_d.

unknown. Figure 5.1 shows the state of heap R as these ten documents are scored with $k = 5$. The first five documents processed (the top row of the figure) enter R and R_k without challenge. Decision steps are required when documents 6 and 7 are scored. In each case, the upper bound for these documents is below the cutoff value (shown as the bold value within the grey bar, marking the limit of R_k). Neither of these two documents enters R (or R_k). On the other hand, when document 8 is scored it is inserted into both R and R_k, and document 1 is deleted from R_k to bring it back to k items. That deletion means in turn that document 0 is now the last in R_k, and that the threshold t should increase from 0.31 to 0.33, a change which causes the removal of document 5 from R. Finally, document 9 is added to R, but does not warrant a top-5 position in R_k.

d	q_1	q_2	q'_2	s_d	rank
0	0.32	0.10	[0.01,0.25]	0.42	4
1	0.30	0.01	[0.01,0.25]	0.31	6
2	0.42	0.08	[0.01,0.37]	0.50	1
3	0.36	0.21	[0.01,0.28]	0.57	0
4	0.39	0.09	[0.01,0.30]	0.48	2=
5	0.21	0.01	[0.01,0.11]	0.22	8
6	0.14	0.10	[0.01,0.10]	0.24	7
7	0.06	0.01	[0.01,0.05]	0.07	9
8	0.37	0.11	[0.01,0.29]	0.48	2=
9	0.23	0.14	[0.01,0.20]	0.37	5

Table 5.1. Example set of documents.

5.1.1 Inter-Term Dependency Analysis

When MaxScore is being used on queries containing nothing but stored terms, the upper bound estimates are independent. But when synthetic terms are also present in the query, the bounds computed by UBnd interact, since if a term is not present, neither can any synthetic term which has it as a component.

The conventional approach to phrase and proximity querying – of constructing a temporary postings list for each synthetic term and then passing them into the query evaluation process – makes it complex to exploit these relationships. But in the approach used in Maxscore, interactions between query terms can be exploited. Consider the original query `"new york city"`. The Sequential Dependence Model expands this query by adding od-1 and uw-8 components for both `"new york"` and `"york city"`. For any document being scored, if the term `york` does not occur in that document, then neither can any synthetic terms using `york`. As a simple example, suppose the maximum possible score for `york` is 6, and the corresponding

Algorithm 8 Dependency-aware variation of function SETSENTINELS. The relative impact of each term's dependencies are reflected in the importance of that term for selection.

SETSENTINELSDEPENDENT(Q, U_Σ, t)

```
 1  s = U_Σ
 2  i = 1
 3  QD = Q
 4  while s > t and QD is not empty
 5       q = first element in QD
 6       QD = QD − {q}
 7       i = i + 1
 8       s = s − UBnd(q, I) + LBnd(q, I)
 9       foreach q_s ∈ QD such that q_s depends on q
10            s = s − UBnd(q_s, I) + LBnd(q_s, I)
11            QD = QD − {q_s}
12  return Q[1 . . (i − 1)]
```

maximum scores for `"new york"` and `"york city"` are 4 and 2, respectively. Assume the minimum for all components is 0.

Step 8 of Algorithm 2 sets $s_d = U_\Sigma$ as the upper-bound score for the document being worked on, document d. If d does not contain `york`, SCORE(d, `york`, I) returns 0, and we just subtract UBND(`york`, I). We can immediately leverage that SCORE(d, `"new york"`, I) and SCORE(d, `"york city"`, I) will also return 0 by also subtracting UBND(`"new york"`, I) and UBND(`"york city"`, I). The resulting drop in potential when `york` is absent is now 12 rather than just 6, because the absence of the one word means that its dependent phrases will also be absent. This larger drop means it is much more likely that documents will be pruned from consideration.

To implement this variation of the Maxscore algorithm, we modify the SetSentinels routine so that it includes dependent synthetic terms in the calculation of the impact on the score of the term's absence. Algorithm 8 shows this modified version.

5.1.2 Generating Approximations

Functions EstMax and EstMin introduce intervals into the scoring computation, and FinalizeScores resolves them. Recall that our goal is to delay expensive operations on the postings lists until it is clear they are needed. That means we want estimators that are fast and that depend only on local information. For example, most scoring algorithms require calculating the number of documents in which a synthetic term occurs – for example, getting the document frequency of `"new york"`. Finding that count requires scanning both stored terms' postings lists to find the intersection. Only then can the scoring begin.

Instead, Algorithm 7's estimator functions return possible scores using statistics at hand, possibly accumulating global information along the way. When all lists have been processed, the counts are known and FinalizeScores can calculate the correct score for any documents that might end up in the set R. For all other documents that did not end up in R, the work of finalizing scores is avoided.

There is, of course, tension between the estimation and finalization phases, since work done in the former has the potential to reduce the time taken by the latter stage, and vice versa. In many cases, the savings from traversing the lists only once more than makes up for the time needed to finalize the scores.

Two major index design decisions affect the quality of the estimates mentioned above. The first is whether or not the index stores term positions at all. If a positional

index is available, then in addition to the document ids and term frequencies we may have easy access to term positions as well. In the more compact document-level index, only term scores are stored. That means that the frequency of each stored term is not necessarily available, having been subsumed into the stored score. As a result, estimating the frequency of a synthetic term is made more complicated. Table 5.2(b) shows the estimated values in this case. The count of times the synthetic term occurs is as few as zero and is bounded above by the component stored term (one of q_j s) that occurs least frequently. (Assuming that the document frequencies of stored terms are available.) Similarly, the number of documents containing the synthetic term is bounded by zero below and by the lowest document count of a component term – for example, `"new york"` cannot occur more often than the less frequent of `new` and `york`.

The second design option is the organization of information in the index. If the index stores both document ids and (in adjacent or interleaved blocks) term positions within each document, the number of times each synthetic term occurs in d can be counted while that document is being scored. If the lists are long, we could estimate that count using the counts of its components: if `new` occurs 20 times but `york` occurs only 4 times, then the bounds on the count of `"new york"` are 0 and 4.

In addition, if the total number of occurrences of the synthetic term that have been encountered so far is also tracked, it is possible to derive an underestimate of the document frequency of the term (and hence an over-estimate of the term's score). Table 5.2(a) shows synthetic term q identified as occurring $count_{q,d}$ times in document d and also shows an array $freq[q]$ that stores the observed collection

frequency of term q across the documents processed so far in the Daat sequence. The $freq[q]$ value is a lower bound of q's count (if it does not occur any more) and the upper bound is that plus $rem(d)$, the number of documents remaining to be scored after this one in the shortest stored-term postings list. Both estimates are safe, in that they encompass a range that necessarily includes the actual value.

In a wide range of similarity computations the critical factors are the two values estimated in Table 5.2. The listed upper and lower bounds can be used to derive upper and lower bounds on similarity score contributions – with the appropriate choice for each case determined by the factors used in the numerator and denominator of the similarity computation. For example, to obtain an upper bound for a *tf.idf*-type computation, the term frequency upper bound must be used, but the collection frequency *lower* bound is needed (as the score is proportional to its *inverse*); and vice-versa for a lower *tf.idf* bound. The EstMin and EstMax functions typically take on the form of the chosen function for Score. For example, suppose we score stored terms under a Language Model using Dirichlet smoothing:

$$\text{Score}(d, q, I) = \frac{count_{q,d} + (\mu \cdot cf_q)}{\mu + l_d},$$

where l_d is the length of document d and cf_q is the collection frequency of q. Suppose further that only a document-level index is available. Then, based on this Score function, EstMin and EstMax can be constructed as follows:

	Term freq.	Coll. freq.
Min	$count_{q,d}$	$freq[q]$
Max	$count_{q,d}$	$freq[q] + rem_{q,d}$

(a) Positional index.

	Term freq.	Coll. freq.
Min	0	0
Max	$\min_j count_{q_j,d}$	$\min_j df_{q_j,d}$

(b) Document-level index.

Table 5.2. Document and collection upper and lower frequency estimates for synthetic terms in: (a) a positional index, and (b) a document-level index.

$$\textsc{EstMin}(d,q,I) = \frac{0 + (\mu \cdot 0)}{\mu + l_d}$$
$$\textsc{EstMax}(d,q,I) = \frac{\min_j count_{q_j,d} + (\mu \cdot \min_k cf_{q_k})}{\mu + l_d}\,.$$

Then, for any given d, q, and I, it is indeed the case that:

$$\textsc{EstMin}(d,q,I) \leq \textsc{Score}(d,q,I) \leq \textsc{EstMax}(d,q,I)\,.$$

5.1.3 Completing Scoring

Algorithm 9 provides a simple implementation of FinalizeScores. Every entry in R is scored, regardless of final outcome, and the top k scoring results are tracked and ordered in R' during that process. It is assumed that index information is most efficiently accessed in document-number order, and this is why R is sorted by its id component. By this stage of the process R is relatively small, typically a small multiple of k, and sorting costs are relatively minor.

The approach shown in Algorithm 9 is expensive, because each call to SCORE accesses two postings lists. If approximate scores within the known range can be used instead, Algorithm 10 is possible (plus other similar variants that use $r.min$ or $r.max$ alone, rather then their average). Now only the information stored in R is used so the calculation is rapid, and, unlike the method in Algorithm 9, there is no invocation of SCORE to access postings lists and neither Q nor I is used. None of these approximations are score-safe, and if the functions used to estimate the bounds are poor, then effectiveness may be affected to an unacceptable degree.

Other finalization methods might estimate the score differently or gather less expensive statistics to provide better estimates of the final score.

Algorithm 9 Exhaustive approach to finalizing the scores.

FINALIZESCORESEXHAUSTIVE(Q, I, k, R)

1 $R' = \{\}$
2 $t = -\infty$
3 **sort** R by $R[i].id$
4 **for** $i = 1$ **to** $|R|$
5     $s_d = R[i].score$
6   **foreach** synthetic term q
7         $s_d = s_d + \text{SCORE}(R[k].id, q, I)$
8   **if** $s_d > t$
9   INSERT($R', \langle d, s_d \rangle, s_d$)
10   **if** $|R'| > k$
11   DELETE($R', k + 1$)
12             $t = R'[k].score$
13 **return** R'

Algorithm 10 Approximate scoring of deferred synthetic terms.

```
FinalizeScoresApproxAvg(Q, I, k, R)
 1  R' = {}
 2  t = -∞
 3  foreach r ∈ R
 4      s_d = r.score + (r.min + r.max)/2
 5      if s_d > t
 6          Insert(R', ⟨d, s_d⟩, s_d)
 7          if |R'| > k
 8              Delete(R', k + 1)
 9              t = R'[k].score
10  return R'
```

5.2 Experimental Structure

All experiments were conducted on a shared cluster of machines, with each machine containing four 64-bit x86 cores operating at 1.8GHz with a total of 16 GB of shared RAM. Most of our experiments are intended to replicate a memory-only environment, so as to isolate the CPU cost of processing a query and ignore disk accesses. To achieve that objective, 4 GB is allocated to the JVM on a single core on one of these machines, an initial warm-up execution of each query carried out to ensure as much as possible of the necessary data is in memory, then the same query is re-executed five more times and the execution time of each of those evaluations recorded as an elapsed system clock time. Once five executions of a query have been measured, the next query is executed in the same manner.

We use the open source Galago search engine[1] and its standard index implementation. In that index, integer and long values are compressed using vbyte compression. Posting lists are written in a term-interleaved organization, with two tiers of skips inserted. The first (lowest) tier is a list of byte positions, with a skip inserted every 500 postings. In order to keep the byte positions from getting too large, an absolute position is stored every 20 regular skips (that is, every 10,000 postings), with the regular skips stored as values relative to the last absolute position recorded. Skips are recorded for the document, count, and position blocks of each posting list. Additionally, for any position block containing more than 3 positions, the length of the positions block in bytes is inserted at the front of the list. This allows that particular block to be directly skipped if necessary. A brief comparison of runs between Galago and Indri[2] show comparable execution times between the two systems.

We use the query likelihood variation of language models (LM) (Ponte & Croft, 1998) as one of the base data points, where we have high-speed execution for a base amount of retrieval accuracy. We call that Ql. Additionally, we apply MaxScore to that algorithm to create Ql-ms, a faster method that provides the same scores and thus results as Ql. Any model providing lower accuracy than Ql-ms without improving speed is not of interest.

Our higher-accuracy model is the Sequential Dependence Model (Sdm) (Metzler & Croft, 2005), where we sacrifice execution speed for increased accuracy. With slight modification, MaxScore can be employed over Sdm as well, providing another refer-

---

[1] `http://www.lemurproject.org/galago.php`

[2] `http://www.lemurproject.org/indri/`

ence point, Sdm-ms. This provides a second endpoint to the spectrum of methods: any mechanism that is slower than Sdm-ms without improving retrieval effectiveness is not of interest.

When dependency analysis is incorporated (Section 5.1.1) into the model, a -msda suffix is added, reflecting that it includes both MaxScore and the dependency optimization. Because Ql contains no synthetic terms, there are no dependencies, so Ql-msda would be identical to Ql-ms.

In addition, a range of methods that estimate score ranges for synthetic terms and finalizing the scores as needed (end of Section 5.1) have also been tried. Again, those are not meaningful for the Ql approach, but can be applied to both Sdm-ms and Sdm-msda. We provide eight variations for each (a total of 16 runs), denoted as Sdm-ms-X or Sdm-msda-X where X is one of the following:

-/ is an approach that does not actually finalize the scores, taking the top k documents as ranked by the minimum estimated score. That is, the set returned is exactly the heap R_k generated prior to step 29 in Algorithm 7. This approach is provided primarily to show the amount of time that is nominally available to the finalization functions. For example, the difference between the time of Sdm-ms and Sdm-ms-/ is the maximum time that a finalization method can take if deferred scoring is to be more efficient than pre-generation of a temporary postings list for each synthetic term.

-2pass finalizes scores by returning to the posting lists for each synthetic term, calculating the actual scores for synthetic terms on a per-document and "as

required" basis. This is the approach shown in Algorithm 9 and is score safe in that it returns the same score that the original sdm approach does.

-ca makes a full pass over the necessary positions in order to gather correct collection statistics, but stores (caches) the count of the synthetic component for the remaining candidates. This cannot be done in the original models because we have no set of potential candidates – that would require that all counts would have to be tabulated before processing. This method is score safe.

-hi uses the estimated high score of the documents to select the top k. Unlike -l, the scores are calculated for all documents stored in heap R, not just the top k stored in R_k. This method is not score safe.

-lo uses the minimum possible score of every document in heap R, not just the top k. This method is not score safe.

-avg estimates as score as the average of its minimum and maximum possible values. This is the method of Algorithm 10 and is not score safe.

-samp estimates collection-level statistics needed in the score from the remaining candidate documents. The statistics are therefore not accurate, but if they are representative of the actual distribution of the entire collection, then the inaccuracy should be small. This is not a score safe method.

-top Reorders R_k using only the completed score, and ignoring the estimates. The set is not guaranteed to be same as ql, as the initial sort was not the same. This is not a score safe method

5.2.1 Collection and Metrics

Experiments were carried out using the queries and documents of the TREC 2006 Terabyte Track (TB06). The collection associated with that track is the 25 million web pages of the GOV2 dataset.

To measure query efficiency, the first 1,000 queries from the "10k" comparative efficiency task of the Terabyte 2006 track were used. Queries were run either as disjunctive keywords (methods ql-*) or with structure added as suggested by the sequential dependence model (methods sdm-*).

Effectiveness results are derived from the 50 manually judged queries from the 2006 adhoc task of the track. Mean average precision (MAP) and precision of the top-ranked 10 documents (P@10) are used to report the effectiveness of runs that are different from the baseline. Runs that are not score-safe are marked with an asterisk in the result tables. Note also that in these experiments we are limited to the set of 50 queries for which relevance judgments are available.

For timing results, we report the mean average time (MAT) per query. As described above, queries are run several times each after an initial "warm-up" run for that query, and the average "hot" processing time is calculated for each query. This methodology gives results equivalent to what could be expected in a system with sufficient memory to retain the entire index in memory, and is representative of typical large-scale processing on very large collections distributed across multiple machines in a cluster. A much larger set of queries is used in these experiments than in the effectiveness measurements, because relevance judgments are not required.

To establish significance of different averages, a one-sided paired permutation test is employed, as described by Efron and Tibshirani (Efron & Tibshirani, 1993). For a given model, we concatenate all of its raw samples together in query order, and swap the ordered pairs during the significance test. We report significance for $p < 0.05$.

5.3 Results

Retrieval Effectiveness

Table 5.3 shows the two measured effectiveness scores for each of the approaches that were explored. At the top of the table, the entries for Ql and Sdm show the effectiveness gain generated by the Sequential Dependence Model. Both of these techniques can be regarded as setting baselines for safe-to-k assessments, but in these experiments Sdm has a clear edge in terms of retrieval quality.

The various approximations listed in the previous section have also been measured. In general – and, in some ways, as a further validation of the Sequential Dependence Model – the more "approximate" the use of the phrase and proximity components, the greater the loss of effectiveness relative to the baseline established by the Sdm approach. For example, the two -avg methods (as per Algorithm 10) score each document according to the mid-point of its possible score range; and both attain effectiveness comparable to the unigrams-only Ql implementation.

It is worth nothing that the Sdm-ms-samp and Sdm-msda-samp methods appeared to suffer no loss in retrieval effectiveness, despite not being score-safe approaches.

Model	MAP	P@10
sdm-ms	0.2207	0.5940
ql-ms	0.1976	**0.5300**
sdm-ms-/*	**0.1900**	**0.5260**
sdm-ms-avg*	0.2021	0.5420
sdm-ms-hi*	0.2065	0.5480
sdm-ms-lo*	**0.1855**	**0.5040**
sdm-ms-samp*	0.2207	0.5940
sdm-ms-top*	**0.1900**	**0.5260**
sdm-msda-/*	**0.1966**	**0.5280**
sdm-msda-avg*	0.2021	0.5420
sdm-msda-hi*	0.2065	0.5480
sdm-msda-lo*	**0.1852**	**0.5040**
sdm-msda-samp*	0.2207	0.5940
sdm-msda-top*	**0.1966**	**0.5280**

Table 5.3. Effectiveness of retrieval using 50 judged queries from the 2006 TREC Terabyte manual runs, measured using MAP on depth $k = 1{,}000$ rankings, and using P@10. Score-safe methods are not shown. Bold values indicates statistical significance relative to sdm-ms.

Model	MAT	Ratio
sdm-ms	9.67	-
ql	5.01	0.52
ql-ms	1.65	0.17
sdm	23.75	2.46
sdm-msda	9.33	0.96
sdm-ms-/*	8.06	0.83
sdm-ms-2pass	14.05	1.45
sdm-ms-ca	12.87	1.33
sdm-ms-avg*	8.41	0.87
sdm-ms-hi*	8.43	0.87
sdm-ms-lo*	8.41	0.87
sdm-ms-samp*	11.43	1.18
sdm-ms-top*	8.01	0.83
sdm-msda-/*	5.75	0.59
sdm-msda-2pass	11.70	1.21
sdm-msda-ca	10.57	1.09
sdm-msda-avg*	6.14	0.63
sdm-msda-hi*	6.14	0.63
sdm-msda-lo*	6.16	0.64
sdm-msda-samp*	9.20	0.95
sdm-msda-top*	5.78	0.60

Table 5.4. Mean average time (MAT) to evaluate a query, in seconds; and the ratio between that time and the baseline sdm-ms approach. A total of 1,000 queries were used in connection with the 426 GB GOV2 dataset. Labels ending with a * indicate mechanisms that are not score-safe. All relationships against sdm-ms were significant.

Retrieval Efficiency

Query execution times are shown in Table 5.4. In the table, the second column shows the mean average time (MAT) to execute queries using the given model. The

	$\|Q\|$							
	1	2	3	4	5	6	7	8
Number of queries	19	206	326	240	140	45	19	6
sdm-ms	0.12s	1.44s	5.73s	10.42s	17.83s	30.43s	41.14s	61.09s
ql-ms	**0.92**	**0.36**	**0.19**	**0.18**	**0.16**	**0.14**	**0.14**	**0.13**
sdm	1.14	3.02	2.72	2.59	2.44	2.11	2.03	2.09
sdm-msda	1.01	**0.98**	**0.98**	**0.97**	**0.96**	**0.94**	**0.93**	1.00
sdm-ms-/*	**0.98**	1.07	**0.86**	**0.89**	**0.82**	**0.73**	**0.73**	**0.82**
sdm-ms-2pass	1.05	2.32	1.69	1.54	1.39	1.20	1.11	1.08
sdm-ms-ca	**0.99**	2.37	1.59	1.37	1.24	1.08	1.03	1.06
sdm-ms-avg*	1.06	1.34	**0.93**	**0.92**	**0.83**	**0.74**	**0.72**	**0.87**
sdm-ms-hi*	**0.98**	1.31	**0.94**	**0.92**	**0.84**	**0.75**	**0.74**	**0.78**
sdm-ms-lo*	**0.91**	1.33	**0.94**	**0.92**	**0.84**	**0.74**	**0.72**	**0.84**
sdm-ms-samp*	**0.99**	2.22	1.42	1.21	1.10	**0.96**	**0.91**	**0.97**
sdm-ms-top*	1.00	1.02	**0.85**	**0.88**	**0.82**	**0.73**	**0.72**	**0.83**
sdm-msda-/*	**0.91**	**0.69**	**0.70**	**0.59**	**0.58**	**0.55**	**0.48**	**0.51**
sdm-msda-2pass	**0.99**	1.94	1.52	1.21	1.15	1.01	**0.87**	**0.85**
sdm-msda-ca	1.00	2.05	1.44	1.06	1.01	**0.89**	**0.77**	**0.72**
sdm-msda-avg*	**0.93**	**0.97**	**0.79**	**0.62**	**0.60**	**0.55**	**0.50**	**0.50**
sdm-msda-hi*	**0.91**	**0.96**	**0.78**	**0.62**	**0.61**	**0.55**	**0.50**	**0.54**
sdm-msda-lo*	**0.97**	**0.98**	**0.79**	**0.61**	**0.60**	**0.56**	**0.49**	**0.55**
sdm-msda-samp*	**0.89**	1.89	1.29	**0.90**	**0.86**	**0.76**	**0.68**	**0.65**
sdm-msda-top*	**0.90**	**0.70**	**0.70**	**0.58**	**0.59**	**0.54**	**0.51**	**0.54**

Table 5.5. Relative execution times as a ratio of the time taken by the sdm-ms approach, broken down by query length. The numbers in the row labeled sdm-ms are average execution times in seconds across queries with that many stored terms (not counting generated synthetic terms); all other values are ratios relative to those. Lower values indicate faster execution. Numbers in bold represent statistical significance relative to sdm-ms; labels ending with a * indicate mechanisms that are not score-safe.

third column, headed *Ratio*, shows MAT as a ratio relative to the baseline Sdm-ms method, with values below one indicating that the time taken is less than for Sdm-ms.

There are number of observations to be made. The first is that the improved effectiveness achieved by Sdm comes at a cost, and even with MaxScore applied, queries cost nearly 10 seconds on average. Indeed, MaxScore brings considerable benefit to both the Ql and Sdm approaches, and the unenhanced Sdm approach is more than two times slower than Sdm-ms. The Ql-ms mechanism is even faster, because no phrase or proximity operators are associated with this retrieval model. The first few rows of Tables 5.3 and 5.4 thus establish the endpoints of a tradeoff between effectiveness and efficiency – Sdm-ms is better from an effectiveness point of view, but the simpler Ql-ms approach is faster.

A second observation is that the dependency analysis technique described in Section 5.1.1 (Sdm-msda) shaves execution cost by only 4% compared to Sdm-ms. However in comparing the Sdm-ms-* models to their Sdm-msda-* variants, the addition of dependency analysis consistently reduces execution cost by another 20%. Third, on this query set the use of delayed scoring with a score-safe completion technique (i.e. *-ca or *-2pass) results in score-safe execution, but no improvements in efficiency over Sdm-ms. The non-safe methods, however, all show efficiency improvement; the Sdm-msda-samp method in particular shows a minor improvement over Sdm-ms, and as shown in Table 5.3, suffered no effectiveness penalty.

Relationship to Query Length

We hypothesize that delayed scoring methods should scale up more efficiently than Sdm-ms as query length increases, however Table 5.4 masks any effect we may

see due to query length. To investigate, we show the same results, broken down by query length; Table 5.5 categorizes the set of queries used in Table 5.4 according to the number of stored terms in each of them. The first row again shows mean average times for the sdm-ms baseline, with all the other numbers in the table presented as ratios relative to the baseline time for that column. Unsurprisingly, execution time grows considerably as queries get longer: the $2|Q| - 2$ automatically generated synthetic terms of the SDM approach become increasingly expensive to evaluate. The columns of Table 5.5, moving from left to right, suggest that longer queries show more efficiency gains from the deferred score calculation. As before, adding dependency analysis to delayed scoring appears to greatly decrease the execution cost, in some cases reducing costs well over 30%.

To further study the effect of query length on relative performance, we ran all queries from the Efficiency Track with length above 4, resulting in a set of 1,868 queries. We report the average increase in execution time as length increases in Figure 5.4. Only the full execution methods and the first pass of the delayed models are shown, to illustrate the amount of "play" available between the approximation phase and the completion phase. Although most of the approximation methods tested here did not result in the desired behavior in improving both efficiency and effectiveness concurrently, the growing gaps between sdm-ms and sdm-msda-/ suggest that we can find the necessary speed; only a clever method to generate collection statistics is needed to find the necessary effectiveness.

Searching for Larger Queries

The results in the previous section hinted towards relative performance improving as queries increase in length. To investigate this phenomenon further, we use a novel query generation method for entity-oriented search in knowledge bases (Dietz & Dalton, 2013) to create large, nested SDM structures. The authors used a variant of latent concept expansion (Metzler & Croft, 2007) that expands on entities[3] instead of terms. The authors used the collection and 41 of the queries from the TREC 2004 Robust track. Table 5.6 shows the results of executing these queries using our methods.

Effectiveness versus Efficiency

Figure 5.3 crystallizes the earlier comments about efficiency versus effectiveness tradeoffs. By plotting P@10 against query execution cost to depth $k = 10$, and MAP similarly against query execution cost to depth $k = 1{,}000$, a clear sense of viable options emerges. In each of the graphs the two dotted lines delineate the region of interest, starting from the two anchoring baselines: Sdm-ms for effectiveness, and Ql-ms for efficiency. The lower-right region represents the challenge of this research – to provide high effectiveness with low execution times. Several techniques appear to show potential, as they fall within our constraints as described above. Depicted in this way, there is clear room for improvement, however given the promise shown by even relatively simple techniques, we are confident that we can progress even further in this region of interest.

[3] In this case an 'entity' is something with a homepage in Wikipedia

Model	MAT
sdm	34.86
sdm-ms	TBD
sdm-ms-/	15.11
sdm-ms-2pass	27.04
sdm-ms-avg	15.18
sdm-ms-hi	15.05
sdm-ms-ca	33.03
sdm-ms-samp	22.55
sdm-ms-samp-1p	26.89
sdm-msda-/	14.99
sdm-msda-2pass	26.84
sdm-msda-avg	15.11
sdm-msda-hi	14.94
sdm-msda-ca	32.09
sdm-msda-samp	22.46
sdm-msda-samp-1p	26.70

Table 5.6. Mean average time (MAT) to evaluate a query, in seconds. A total of 41 queries were used in connection with the TREC 2004 Robust dataset.

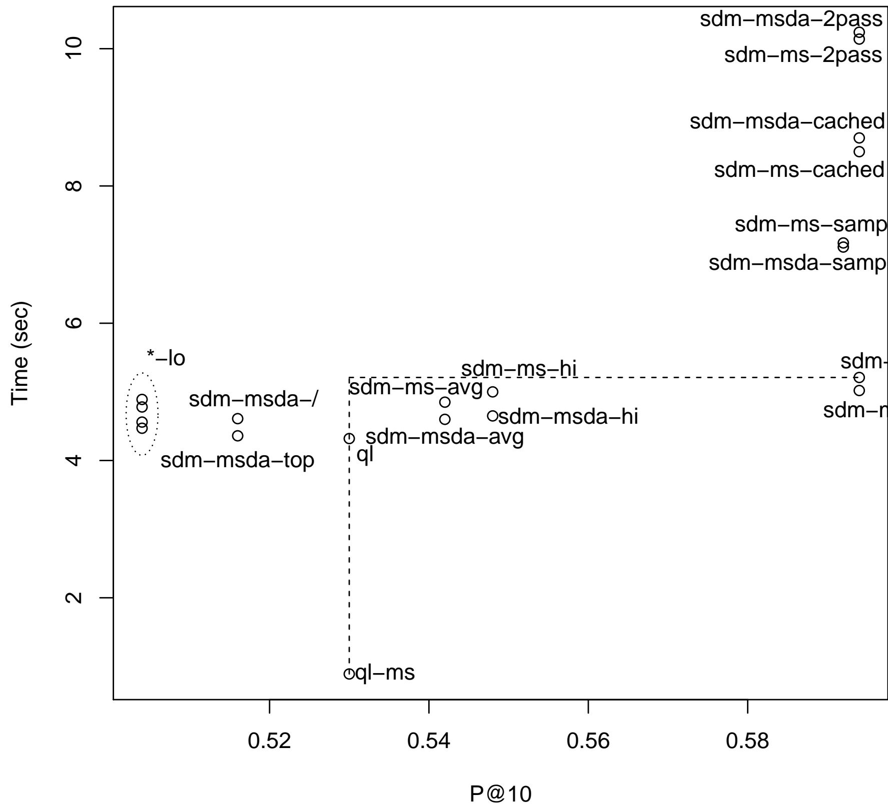

Figure 5.2. Execution time (in seconds) against retrieval effectiveness at depth $k = 10$ with effectiveness measured using P@10. Judgments used are for the TB06 collection, using 50 judged queries.

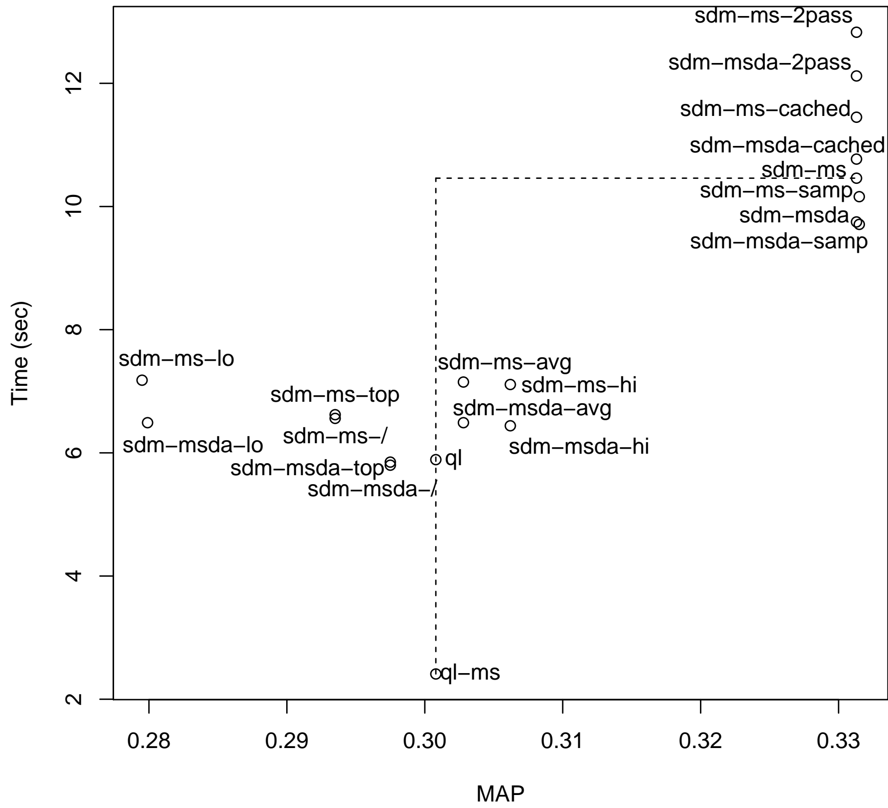

Figure 5.3. Execution time (in seconds) against retrieval effectiveness at depth $k = 1{,}000$ with effectiveness measured using MAP to depth 1,000. Judgments used are for the TB06 collection, using 50 judged queries.

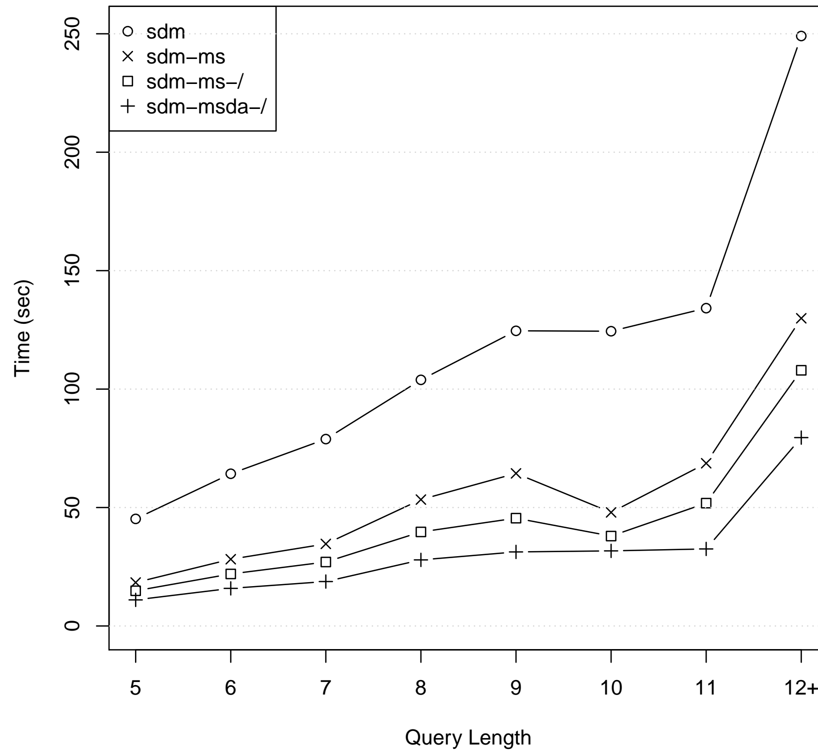

Figure 5.4. Execution time (in seconds) as query length increases. Only queries of length 5 or greater from the 10K queries of the TREC 2006 Terabyte comparative efficiency task query set were used.

Another point of interest is the change in relationship as k goes from 10 to 1000. sdm-ms and sdm-msda increase in execution time much more than all of the delayed scoring methods; this behavior suggests that the relative benefits of delayed scoring increase as more results are requested. Should this behavior extend to other domains, then recall-oriented tasks (e.g. legal search) would greatly benefit from this approach.

CHAPTER 6

A BEHAVIORAL VIEW OF QUERY EXECUTION

The previous chapters introduced three new optimization techniques for structured queries in information retrieval. Each of these techniques is useful in its own right, but each technique was also developed in isolation from the others. A natural consequence of developing these techniques is the desire to have them all available concurrently in a single system. In doing so, we improve coverage over what queries can be more efficiently executed, and in some cases we may see additive benefits from having multiple optimizations applied to a query. In order to benefit from multiple optimizations at once, we must implement them in the system, and recognize when it is appropriate to use them. However this undertaking is not always painless - each optimization is a different code path, and the logic to exercise must be injected into the system at the right place in order to avoid duplicating large sections of code, which can lead to code drift, among other difficulties (Hunt & Thomas, 1999; Fowler, 1999). In addition to the dilemma of combining these methods, we must also balance the need to design and support new retrieval models that were not present when the system was originally built. Given these challenges, what is an effective way to build a system that can adequately meet these needs? This chapter of the thesis focuses on presenting a novel answer to this question: We follow a library- (or toolkit-) based

approach to component design, but we additionally make use of behaviors attached to these components to guide selection of higher-level components, providing convenient places to affect query construction and execution. We implement this design in Julien, a novel retrieval stack library built on the Galago indexing system which leverages the behaviors of the provided components to fluidly incorporate query execution changes. This design allows for a more adaptive approach to query execution than in previous retrieval systems.

We begin with a description of the major components we consider necessary to construct and execute a basic retrieval run over an index using a single query. Based on this high-level model, we proceed with a description of query representation in Julien.

We then proceed to discuss the differences in design between Indri, Galago, and Julien, and conclude with a short example to illustrate how the design of Julien allows for easier extensibility than these previous systems.

Finally, we describe the implementation of each of the three optimizations from this thesis in Julien, and show that the component-oriented design of Julien makes the individual extensions uncomplicated. In addition to the individual implementations, we also discuss how we implement all three optimizations "under one hood", to produce a retrieval system that can make use of all three optimizations concurrently under the appropriate conditions.

6.1 Executing a Query in Julien

Figure 6.1 shows a simple component diagram[1] of the major parts of a retrieval system in Julien. The "lollipop" symbol (small circle on a stick) represents an implemented method or interface, and the "socket" symbol (semicircle on a stick) is a dependency or needed interface. Using these definitions, we can see that the `Index` component provides needed information to the `Operator`s, which when combined with an `Accumulator`, can construct a `QueryProcessor`, which is used to execute the query. We now discuss these components in turn.

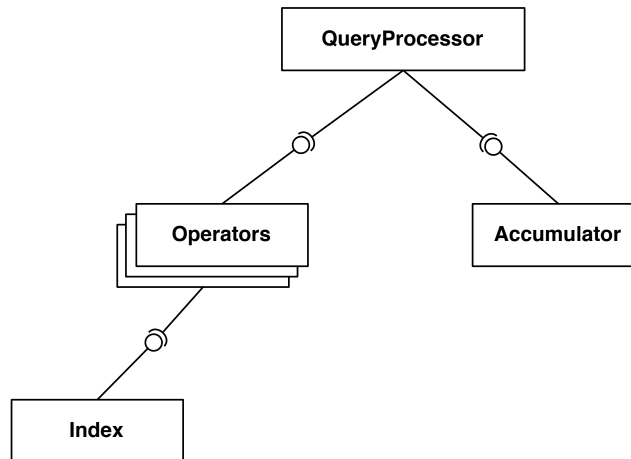

Figure 6.1. A component diagram of the basic parts of Julien.

[1] `http://en.wikipedia.org/wiki/Component_diagram`

The `Index`

In Julien the index is an interface that connects to the information stored in a standard index file, and provides a wide set of methods to support convenient access to basic index key and retrievable statistics, iterators over the underlying vocabulary, and over posting lists of the keys. Currently the index supports accessing index structures built using a lightweight variant of Galago v3.3.

The `Accumulator`

The accumulator is an object that stores results during query evaluation. This allows the logic of final ordering to be encapsulated in a single object, as well as decoupling the ordering logic from the processor code, which is simply meant to score documents as quickly as possible. Note that in some cases, the current state of the accumulator must be accessible to ensure proper operation of the processor - we will discuss this issue further in Section 6.5.1.

The standard accumulator implementation is a priority queue sorted on increasing score (this makes it easy to dequeue the "lowest" document off the queue, and keep the memory footprint small). Other implementations are possible, such as only keeping documents with a score that is within 10% of the maximum score generated, or re-ranking documents based on the other documents in the accumulator.

The `Operators`

There are two subclasses of operator in Julien: *views* and *features*. A view takes an index key and/or an identifier as input, and produces information from the index based on those parameters. A commonly implemented view is a "count" operator:

when given a valid index key and identifier, the operator produces the number of times that index key occurs in the identified retrievable. A feature operator takes in the information provided by one or more views, and maps it to the space of floating point numbers. The output of this mapping, and the outputs of further transformations based on this output, are what the system uses to rank retrievables. In short, we define a retrieval model to be one or more features applied to one or more views from the index. Views provide direct access to information stored in the index, whereas features are functions over the information provided by views.

The `QueryProcessor`

The *query processor* is a module that accepts a query (in the form of operators) and an accumulator which will hold the results, and executes them to produce a ranked list of results. The processor implements the scoring regime (i.e. the process of evaluating the query operator graph against the documents in the index), and serves as the main driver for executing a query against an index. While one can implement the scoring regime by hand, keeping the underlying data structures properly up to date can be error-prone without careful consideration, and by encapsulating this responsibility, we have a convenient place to drop in new optimization strategies for execution when they arise. For example, the ExhaustiveScoreDocuments, Maxscore, and WAND algorithms discussed in Chapter **??** are all implemented as processors in Julien.

To give an idea of what is involved in processing a query in both Galago and in Julien, we provide listings that use the minimal amount of code necessary to run

a single query (Listings 6.1 and 6.2). We begin with a walkthrough of single-query execution in Galago.

Listing 6.1. Executing a basic retrieval stack in Galago.

```
1  /* imported classes */
2
3  val params: Parameters =
4    Parameters.parse(``{ ``index'' : ``./myIndex''}'')
5
6  val parsed: Node =
7    StructuredQuery.parse(``hubble telescope achievements'')
8
9  val retrieval: Retrieval = RetrievalFactory.instance(params)
10
11 val transformed: Node =
12    retrieval.transformQuery(parsed, params)
13
14 val results: QueryResults =
15   retrieval.runQuery(transformed, params)
```

Lines 3-4 construct the parameters object that will be provided to other components through the process. We minimally need the location of the index, which is specified in line 4. Lines 6-7 parse the string form of the query into a tree of `Node` objects that represent an annotatable abstract syntax tree (AST) of the query. The `Node` objects themselves do not provide an API to execute the query (i.e. you cannot ask a `Node` what the score is for document x). After transformation and annotation (lines 11-12), the AST carries enough information to be materialized into an executable query. Line 9 creates a `Retrieval` object, which provides the interface for transforming and executing a parsed query tree. In lines 11-12 the query is transformed and further annotated via sequential application of the traversals in the system. In lines 14-15, the transformed AST, and any necessary parameters (e.g., non-default number of results requested) are passed into the `runQuery` method of the

`Retrieval` object, and results of the query evaluation are returned. The `runQuery` method handles converting the query AST into the composed set of Galago iterator objects that supply the API sufficient to score documents (materializing the query), and executing the materialized query over the index.

Listing 6.2. Executing a basic retrieval stack in Julien.

```
1  /* imported classes */
2
3  val index: Index = Index.disk(``./myIndex'')
4
5  val queryFeatures: Feature =
6    generateQuery(``hubble telescope achievements'', index)
7
8  val acc: Accumulator[ScoredDocument] =
9    DefaultAccumulator[ScoredDocument]()
10
11 val results: QueryResults =
12   QueryProcessor(queryFeatures, acc)
```

Walking through the process in Julien, Line 3 opens the index, which is located at "./myIndex" in this example. After opening the index, we use the user-defined `generateQuery` function to convert the input query string ("hubble telescope acheivements" in this case) into the graph of features that can be processed (materialization). In reality, several predefined functions exist in Julien (e.g. `bow`, `sdm`) to turn the string into the correct structure. The user can in fact take any action they want to generate a feature for the query processor on line 6 - this particular example simply assumes an existing predefined function `generateQuery` to act as a placeholder for a real implementation. Listing 6.3 shows an hardcoded version of this function, tailored for the query used in this example. At this point, the query features supply the API sufficient for document scoring (e.g., `queryFeatures.eval(5)` would return the score of evaluating the query against the document with internal id 5).

After opening the index and building the query, we create an accumulator of `ScoredDocument` objects (lines 8-9). A `ScoredDocument` is the standard retrievable representation in Julien, although more types can be defined (we will discuss such an extension in Section 6.5.3). Finally, we pass the constructed query and accumulator to the `QueryProcessor` (lines 11-12), which executes the query and returns a ranked set of results.

Although neither approach requires a significant amount of code to execute a query (assuming the `generateQuery` method substituted in is built-in), the two approaches expose different parts of the query execution process. Galago exposes the AST representation of the supplied query, which allows the user direct manipulation over the AST, and provides an opportunity for any changes to the tree that require query-level information (see Section6.6 for discussion on this point). Julien places the burden of actually materializing the query on the user. This makes query-level inference difficult, but it removes the indirection caused by using an AST between specifying the query textually and executing the query in the system. Galago performs a large number of steps internal via the `runQuery` method. Actual materialization of the AST occurs inside this method, therefore the user never has direct access to an object that provides an API that can be used to execute a query.

Both approaches provide ways to extend the set of available operators. In Galago, a new operator can be specified by adding the textual description (e.g., `#combine`) and the fully-qualified iterator class (e.g., `myiter.school.edu.CombineIterator`) as sub-parameters to the `runQuery` method. Of course, in addition to adding the parameters, the iterator must be implemented and be visible on the classpath of the

JVM. In Julien, a new operator can be added by implementing the new operator code (which must also be visible to the JVM), and using it in the materialization code (i.e., `generateQuery`).

Both systems also provide a way to affect the execution process. Julien requires the user to either let the library select a processor to execute the query (lines 11-12 of Listing 6.2), or to explicitly construct a processor to handle the query. Galago will by default select a processor for you internally, but if a particular processor is desired, it can be specified via the parameters passed into the `runQuery` method (the `params` variable in Listing 6.1).

Neither approach is clearly advantageous over the other. Galago automates the majority of the materialization and execution process, whereas Julien makes these steps much more explicit to the user. The choice of the "right" toolkit to use depends heavily on what aspect of retrieval the user plans to modify.

6.2 Representing a Query

Out of the components shown in Figure 6.1, the operators are the most complex, as they are the objects that comprise the functional instantiation of the query. Therefore we cover this aspect of the library in more detail here.

Queries are implemented as directed acyclic graphs (DAGs) with a terminal (or sink) node that when evaluated, produces the score for a document. This representation is much like the Inference Network model (Turtle & Croft, 1991), but we drop the requirement that the resulting function be a probabilistic model.

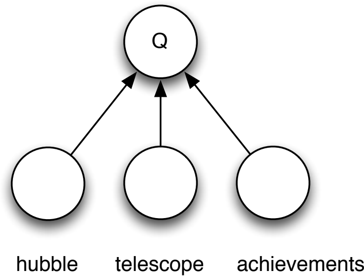

Figure 6.2. A simple query tree.

We assume that a node may represent any user-definable operation (e.g. generating counts from an on-disk posting list, combining scores, smoothing, etc.). We use query 307 from the TREC 2005 Robust track ("hubble telescope achievements") as an example query. Figure 6.2 provides the simplest possible graph representation of this query. The sink (labeled Q) constitutes the feature that produces the final score for a given document. The views, which are all leaves, each represent the source of count information from the index for their labeled query terms. This count information is combined at the sink node to produce a final score for a given document. Note that most simple keyword models, such as the Language Model or BM25, actually implement scoring (the translation of a raw statistic, such as a count, into some kind of belief about relevance) at the term level. As in many research IR systems, in Julien the scores are calculated at retrieval time, based on the raw term statistics which are stored in the index. Therefore, this same model, materialized in Julien, would involve feature nodes that first transform the raw counts produced by the views into scores, which are then "passed" up to the combining feature. This

explicit construction is shown graphically in Figure 6.3, and in code (creating a language model of the query) in Listing 6.3. The additional term-level features typically clutter the query graph representation, so for the remainder of this chapter we assume they are present unless stated otherwise. In the case of the code listing, we can automate the repeated construction of term-level nodes, and generalize to create a suitable incarnation of the `generateQuery` function of Listing 6.2.

Listing 6.3. Manually specifying a query as a language model for Julien.

```
1  /* imported classes */
2
3  val root =
4    Combine(
5      Dirichlet(Term(``hubble''), IndexLengths()),
6      Dirichlet(Term(``telescope''), IndexLengths()),
7      Dirichlet(Term(``achievements''), IndexLengths()),
8    )
```

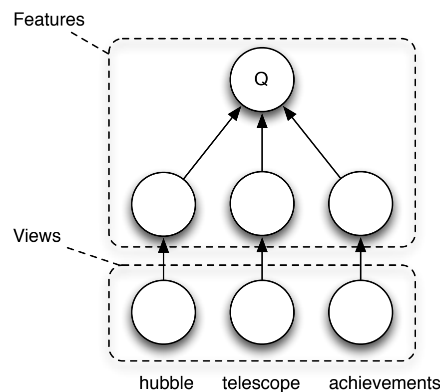

Figure 6.3. A simple query tree, with both features and views shown.

6.3 The Behavioral Approach

Now that we have a basic understanding of formulating and processing a query, we introduce the idea of *behaviors* that can be attached to operators within the query, allowing for informed optimization throughout query execution. We begin by introducing the Theory of Affordance, which serves as the underpinning of behaviors in Julien.

6.3.1 Theory of Affordances

The idea of an *behavior* was first presented by Gibson as early as 1977, and originally referred to latent "action possibilities" that exist in the environment independent of an actor, however always defined in relation to an actor (Gibson, 1977). For example, a heavy wooden chest with handles on the side has an affordance of lifting and carrying, but only to large adults capable of lifting it. The affordance is not present for smaller individuals, such as toddlers or infants. The view presented by Gibson is the currently accepted interpretation in cognitive psychology.

In 1988, Norman modified the idea of affordances to only refer to the set of action possibilities that were readily perceivable by an actor (Norman, 2002). Since then, Norman's definition has seen significant use in both the Computer-Human Interaction (CHI or HCI) community (Gaver, 1991; McGrenere & Ho, 2000; Dalgarno & Lee, 2010), as well as robotics (Fagg & Arbib, 1998; Stoytchev, 2005a, 2005b; Sen, Sherrick, Ruiken, & Grupen, 2011). We use this definition as the paradigm for component design in Julien. In this case, the agent that takes action is a query processor, which actually manipulates and executes queries. Based on the behaviors

present on the operators in the query, we can select the correct processor to execute the query as efficiently as possible.

Consider Figure 6.4, where we show a hypothetical query consisting of 4 view operators (nodes A-D), and 4 feature operators (nodes 1-4). We also have 3 behaviors exhibited to varying degrees by the operators. The behaviors are indicated by colored shapes; in Figure 6.4, all of the operators except operator 2 exhibit one or more behaviors. For example, all of the views are loaded into memory, as indicated by the red triangle behavior. Several other behaviors are exhibited by the features.

The purpose of exposing these different behaviors is to inform the QueryProcessor what each operator is capable of, thereby allowing the QueryProcessor to select the most aggressive execution strategy (encapsulated as a Processor instance) given the exhibited behaviors. For example, in most systems, retrieving an entire document from the document store is an expensive operation. In the example shown by Figure 6.4, knowing that nodes 1 and 3 require document text suggests that we can save significant time by delaying the execution of those features during evaluation. A capable processor (such as the ones discussed in Chapter 5) may evaluate nodes 2 and 4 first, and only evaluate the remaining features if the partially completed score warrants it. This simple modification to execution may save significant computational cost when processing queries with operators that exhibit the green diamond behavior.

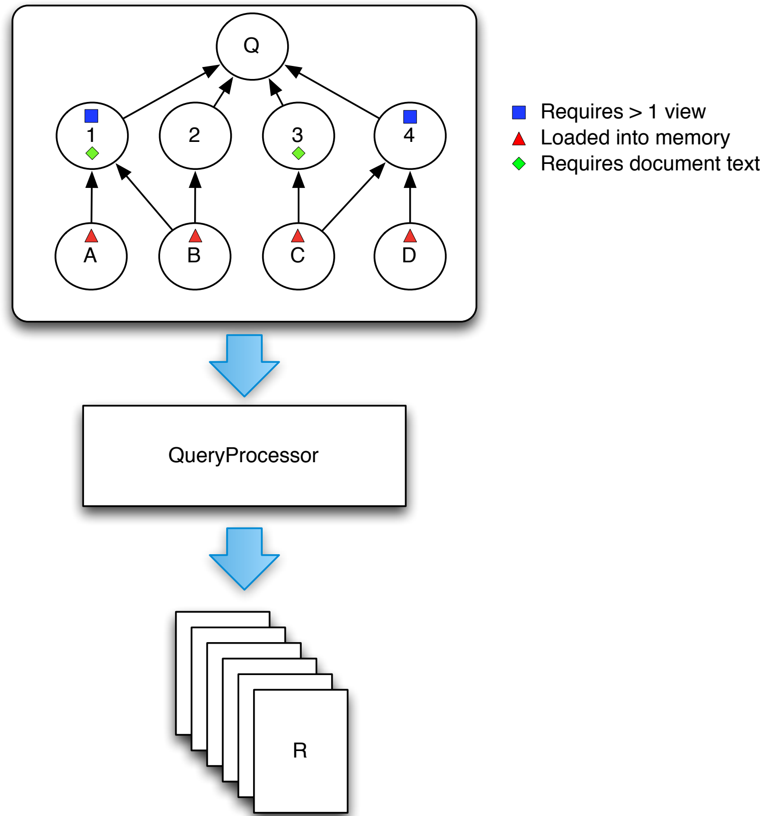

Figure 6.4. A query Q, with operators exposing different behaviors, is passed to the QueryProcessor, which executes the query and produces a result list R of retrievables.

6.3.2 Built-In Behaviors

We now discuss several of the different behaviors currently implemented in the Julien system. As we describe each behavior, we also provide examples of how the system may alter its operation based on these indicated behaviors.

Bounded. An operator that makes available upper and lower bounds for the values it provides. This provides essential information for pruning algorithms such

as Maxscore and WAND that rely on upper and lower bound estimates of each scorer. An operator with this behavior would serves as valid input to the LBnd and UBnd functions used earlier in the thesis. An operator that does not explicitly exhibit this behavior, however, would be ineligible, and the system would avoid using Maxscore and WAND to process the a query using that operator.

Finite. A trait to indicate that the operator has finite length (meaning it provides non-default values for a finite set of retrievables). Additionally this trait indicates whether the operator is dense (provides values for all retrievables) or not. Since we have defined all collections as finite, all built-in views exhibit this trait. Future operators may not exhibit this behavior (for example, an operator that estimates the current "hotness" of a search term based on a microblog stream. The operator may not have a constant value, but it also is never "done" iterating), therefore the system should exclude non-Finite operators when determining when to cease iteration over the collection.

Movable. An extension of the Finite trait. Operators that require some updating of internal state when a value based on a retrievable key is requested (e.g. moving a disk-based iterator that underlies a view). Typically only views will exhibit this trait. While all processors need to be aware of operators that are Movable, every operator that is *not* Movable (perhaps because it is backed by an in-memory hash table) can be ignored by the processor when updating state, reducing the overhead incurred by removing needless iteration over an object that does not require an update call.

Conjunction. An operator with children that requires all children to have non-default values in order to produce its own non-default value for a given identifier. This

indicates that if the children are not shared among other nodes, then any Movable operators can be moved aggressively (i.e. line up to the maximum candidate of all children, instead of the minimum candidate). The synthetic components discussed in Chapter 5 were all term conjunction operators, so they all exhibited this behavior.

Random Access. This trait indicates that an operator implementation has fast random-access capabilities. This means next, seek, and random-access operations are all O(1). If this trait is not specified, the operator is assumed to have fast "next" lookup, moderate seek cost, and high random-access cost. Typically a view will exhibit this or the Movable behavior.

NeedsPreparing. An operator with this trait requires an initial pass over the index before it can correctly score retrievables. If a query contains operators with this trait, a preliminary pass is performed to prepare these operators for scoring. Currently, the implemented conjunction view operators (e.g. the view of the phrase "`new york`") express this trait. The trait is placed here in order for the view to be able to properly respond to requests about statistics of the conjuction it represents.

The system is designed to easily add new components as well as new behaviors. By following a library-oriented approach, an existing retrieval system construction can be easily extended by replacing one or more of the components shown in Figure 6.1. By incorporating behaviors into the component design, we also allow the system to continue to leverage previously implemented improvements by representing components via their behaviors. This balancing of extensibility and efficiency may sound straightforward, and it is an important and sometimes elusive aspect of an extensible software system.

6.4 Example: Generating New Smoothing Statistics

We now have component-oriented design and the Theory of Affordances as the central concepts that drive the design of the core Julien system, as well as any extensions to add functionality to the system. Let us now extend Indri, Galago, and Julien with some new functionality, to explore the difficulty of extending each system.

Let the proposed extension be the following: We would like to add a new feature to score an extent probability (which can be a unigram, or a span of terms such as an ordered distance) as

$$P_{lowpass}(t \mid d) = P(\text{t occurs exactly once in d}) * P_{ml}(t \mid d) \tag{6.1}$$

We can think of $P_{lowpass}(t \mid d)$ as the a weighted probability of t occurring given d (the $P_{ml}(t \mid d)$ term), where the raw probability is weighted by some belief about the importance of t (the $P(\text{t occurs exactly once in d})$ term).

All three systems already have similar operations, such as the Dirichlet smoothing function from the language modeling framework. In order to properly score the extent, however, the new scoring function requires the number of documents in the collection and number of documents where the extent occurs exactly once (i.e. tf = 1). Using the definitions from earlier in the document, let us call the function the "lowpass" function, defined as follows:

$$lowpass(d, t, I) = \frac{|\varsigma(t)| + 1}{|D|} * \frac{\textsc{Count}(d, t, I)}{|d|}$$

where

$$\varsigma(t) = \{d \mid t \text{ occurs in } d \text{ exactly once } \}$$

We have two steps to complete this extension: 1) implement a new scoring function, and 2) implement a mechanism to collect the correct statistics for this new scoring function. We assume that we do not re-index the collection to statically store the statistic, as we first would like to determine if using this operator is worth rebuilding indexes to store the statistic. We now discuss how one might perform these steps in Indri, Galago, and Julien, and discuss the various benefits and drawbacks of implementing this extension in each code base.

6.4.1 Extending Indri

Indri was originally designed as an intersection of the language modeling framework and the inference network. As such, it implements these built-in functions very well, and executes them quickly. However, the design focus was on a tightly integrated system, and not on extensibility. A reasonable way to estimate the difficulty of performing this extension is to look at a prior extension by (Ogilvie & Callan, 2005), who performed an extension that is similar to step 1 in this extension (Fisher, 2013). The work conducted was to investigate hierarchical language models in XML (Ogilvie & Callan, 2005), which involved adding a "shrinkage" operator to the Indri system. A review of the commit logs and the author documentation around this time indicate that approximately 6 classes were substantively edited or added to implement the shrinkage operator. If we assume that similar changes would be needed to implement the lowpass operator, the commit logs indicate that step 1 of

this extension would involve adding a new operator class as well as modifying several classes the operator would interact with, and possibly adjusting the grammar of Indri's query parser in order to accept queries involving the lowpass operator.

Implementing step 2 would involve modification to an entirely separate logical part of the codebase, which is involved in gathering statistics to annotate query nodes with the correct information. The typical process is to copy the nodes requiring statistics into a new subtree, traverse the index with only that subtree, collecting statistics as necessary for each component, then annotating the original tree with the gathered statistics. Currently, the module responsible for accumulating statistics will gather collection frequency (cf), document frequency (df), collection size ($|D|$), and the number of tokens in the collection ($|C|$). Unfortunately that only fulfills half of our need, so we need to modify the accumulator to have space for the statistic we need, the logic when adding statistics to recognize when to update the new statistic, and the logic to annotate the new operator node with the correct statistic after the statistics pass. Given the design of Indri, these three mechanisms are implemented in three different classes.

After implementation, the overall efficiency of the system has also been affected. While the new operator may now work, the accumulator must now track this new statistic for every node that requires a statistics-gathering pass, even if the node does not use that particular statistic. Additionally, the only way to effectively implement

the changes is to recompile the source code of Indri[2]. All told, such an extension is clearly nontrivial in Indri, although certainly possible.

6.4.2 Extending Galago

Galago was designed to allow for more flexibility than Indri (Strohman, 2007); in particular Galago provides two mechanisms for extension: 1) traversals, which are tree-walking operations, and 2) operators, which are similar to the operator definition in Julien, but require registration with a factory class to recognize the operator pattern. The traversal subsystem is similar to Indri: a sequence of traversals is applied to the query tree in a particular order. The operators work by defining a tag in the query language (e.g. `#combine`); a query comes in as a string, and as tags are recognized, they are converted into Node objects, which contain the information necessary to create an iterator. The nodes can then be annotated by external parameters or traversals, and then the object that actually traverses the index is fully instantiated using the annotated query tree, and scoring takes place.

Given these mechanisms, implementing the new statistics and operator is not as complex as in Indri. The entry points even allow for implementing the new logic in a separate project, and simply including them as a dependency to Galago via the classpath and parameters to the Galago executable.

Step 1 requires a new Iterator class that implements the scoring logic of the new operator, similar to the one of the built in Iterators. The iterator can then be

---

[2]While linking statically to the Indri libraries with the new code is possible, most of the automation in servicing a query would have to be duplicated in order to use the new classes. Such an approach would be wasteful and error-prone, so we do not consider such an approach here.

associated with an operator tag (e.g. `#feature:lowpass`) which will be transformed into the iterator when the operator tag is seen.

Step 2 can be implemented by creating a new Traversal class that, when it encounters a class that requires the new statistic, can perform a pass over the index to collect that statistic and annotate the operator node.

Both of these new components can be registered with Galago via the parameters that are passed into the system. As long as the classpath is properly configured, this functionality can be added without recompiling the base code of Galago. This is a significant improvement over the static definitions found in most of Indri. In terms of performance, the extension has a smaller memory footprint on the system than in Indri. The traversal will only trigger an index pass for the nodes that require the new statistic, instead of tracking the statistic for every node (whether the node uses the statistic or not) as in Indri.

Both Indri and Galago, which are built as frameworks, impose limitations on the way in which the extension can be implemented. The logic used to prepare an operator is divorced from where the prepared information will actually be used. Both Indri and Galago require the code to gather statistics to be in code for traversals over the query tree. The query tree is annotated with statistics, and then the proper index iteration structures are created with the annotations. This means that correctness checking (e.g. ensuring that $\varsigma(t) \leq |D|$) is difficult to enforce. Whether correctness verification is done at the traversal where the statistic is gathered, or it is done at the operator where the statistic is used, there is a section of the code path (between gathering and using the statistic) that cannot be always accounted for, and can

introduce bugs that are difficult to isolate. This can happen if any traversal later inserted after this one interferes with the value of $\varsigma(t)$ before the operator can read it. Correctness of $\varsigma(t)$ can be checked at both places, but that produces two places that must be maintained if the conditions for correctness ever change.

The traversal system in Galago, while powerful, also creates a chain of dependencies: earlier traversals may make transformations that later traversals depend on. The transformations and their dependencies are difficult to document, and even more difficult to formalize to ensure correctness. In Indri this system was not often modified, so this dependency chain could be verified once and that would be sufficient. In Galago, the traversal sequence was *meant* to be modified, but the dependency chain is difficult to specify. The solution Galago presents is to allow traversal insertion in one of three places: before, after, or as a replacement of the entire built-in traversal sequence. This skirts the issue for traversals that do not interact with any of the others in the sequence, but if the traversal needs insertion at a certain point in the sequence, recompilation of the base code is required.

6.4.3 Extending Julien

Julien is designed around behaviors, so instead of looking at the two steps as operations, we should determine if any of them are behaviors we already have in the system. Although we need a new statistic, what needs to happen is that the operator needs access to a preliminary pass where statistics can be gathered. This is precisely what the `NeedsPreparing` behavior, defined above, encapsulates. Therefore, we only need to add a new feature that exhibits `NeedsPreparing`, and the feature will have access to an initial pass where it can properly collect the statistics needed.

The `NeedsPreparing` behavior is implemented in Julien as follows[3]:

Listing 6.4. The `NeedsPreparing` trait.

```
1 trait NeedsPreparing {
2   protected var amIReady: Boolean = false
3
4   def updateStatistics(docid: InternalId): Unit
5   def isPrepared: Boolean = amIReady
6   def prepared: Unit = amIReady = true
7 }
```

The `updateStatistics` method is the only method to implement for our operator; it tells the implementing operator to update its information on the provided document id. The `isPrepared` method reports whether the operator is prepared, therefore not requiring a preparation run. The `prepared` method is called when the preparing run is complete, in case the operator needs to perform calculation over the total of the statistics. An example implementation of the new operator in Scala is shown in Listing 6.5.

[3]The implementation language is Scala.

Listing 6.5. Example code for the example LowPass Operator in Julien

```
1  class LowPassOperator(
2    term: CountStatsView,
3    lengths: LengthsView
4  )
5   extends Feature
6   with NeedsPreparing {
7
8    var singletons: Int = 0
9
10   def updateStatistics(docid: InternalId): Unit {
11     val c = term.count(docid)
12     if (c == 1) singletons += 1
13   }
14
15   lazy val factor: Double = {
16     if (isPrepared) {
17       val numDocs = term.statistics.numDocs
18       if (numDocs < singletons)
19         throw new Exception(``Bad value.'')
20       return (singletons + 1).toDouble / numDocs
21     else
22       throw new Exception(``I was not initialized.'')
23   }
24
25   def eval(id: InternalId): Double = {
26     val tf = term.count(id).toDouble / lengths.length(id)
27     return factor * tf
28   }
29 }
```

The constructor is defined as part of the class declaration, on lines 1-4 in Listing 6.5. The `term` argument is of type `CountStatsView`, which is a type that provides per-document counts as well as global statistics (e.g. collection count, document count) about a given term. The second argument, `lengths`, is of type `LengthsView`, which is a view that provides the lengths of documents. The `singletons` member initialized at line 8 is the counter variable that, after updating, should equal ς(`term`).

The `updateStatistics` method (lines 10-13) collects the count for the given `docid`, and if the count is 1, updates the `singletons` variable. The `factor` member (lines 15-23) is actually a constant (hence the `val` modifier), but it is calculated lazily (hence the `lazy` modifier). Therefore, its value is not set until it is first requested. At that first request, the method the member is attached to is executed, and that value is then set to that member. We perform a check at this calculation to make sure that `singleton` is in fact up to date (line 16), and it not, throw an exception to indicate it was not set. We also perform our correctness check at line 18. If the check passes, we make the calculation for `factor` once making use of the `numDocs` value provided by the `term` object, and set this value once. Finally, we have the `eval` method on lines 25-28. The method is straightforward, using the count and length provided by the views, and returns the lowpass value for that term/document pair.

Like Galago, if the extension code is outside the base code of Julien, then it must be specified in the classpath. Unlike Indri and Galago, the logic of the needed statistic is contained in the class that also uses the statistic. The behavior communicates only what is needed for the underlying system to use it. In this case, the `Lowpass` operator exhibits the `NeedsPreparing` behavior, telling the query processor that a preliminary run is necessary before proper scoring can begin.

While we only needed to add an operator that uses a pre-existing behavior, adding a new behavior would have only involved creating the behavior class and then adding the logic to make use of the behavior (in this case the first pass allowing for operator preparation).

Recap

We discussed extending the Indri, Galago, and Julien retrieval systems to add a new operator that made use of a new statistic gathered from a first pass of the collection. Despite having similar operators and statistics, implementing the extension in Indri requires modification across numerous class files, as well as recompilation of the base code. Galago is a marked improvement, requiring modification of fewer files, but there is still some uncertainty towards how to properly insert the traversal, and the logic is still split over multiple classes.

Julien is a further improvement - by leveraging shared behavior between operators, only a single new class was needed to implement the extension, and all of the logic was self-contained. As we will see, not all modifications are so simple as this example, therefore limiting extensions to one file will not be possible. However we will still be able to make modifications to the core system by simply pulling in the core classes and extending them as needed.

6.5 Implementing Multiple Optimizations Concurrently

We now look closer at the implementation details of the optimizations dicussed in Chapters3, 4, and 5, specfically how they can simultaneously fit together into Julien.

6.5.1 Implementing Query Flattening in Julien

We now discuss the approach taken to implement a generic form of the flattening optimization, introduced in Chapter 3, in Julien. We implement this optimization by defining the **Distributive** behavior. The purpose of this behavior is to indicate that

the implementing operator is an interpolated subquery, and if the operator occurs in the correct context, that this subquery's operator can be removed. An operator that exhibits this behavior has two new methods exposed:

1. `distribute()` $\rightarrow$ `List[Feature]`

2. `setChildren(List[Feature])`

Given a Feature node N, and its children $c_1, \ldots, c_j$, the `distribute` method reweights all c_i by multiplying the weight of N into each child, then returns the list of reweighted child nodes. The `setChildren` function replaces the current set of children with a new provided set of child nodes. These two methods alone are not enough to cause the query graph to flatten under the right circumstances; since this optimization involves changing query structure we can add logic to recognize and collapse the correct interpolated query substructure before sending it to be processed. Such a transformation is typically referred to as a *query rewrite*. In Julien constructing a retrieval pipeline that uses one or more rewriters is no more difficult than adding a function call after initially building the query. Consider Listing 6.6, which shows the basic retrieval stack augmented with a query rewriter called `Flattener`, which performs the desired transformation.

Listing 6.6. Executing a basic retrieval stack in Julien with a simple rewrite.

```
1  /* imported classes */
2
3  val index: Index = Index.disk(``./myIndex'')
4
5  val queryFeatures: Feature =
6    generateQuery(``2014 winter olympics'', index)
7
8  val rewritten: Feature = Flattener.rewrite(queryFeatures)
9
10 val acc: Accumulator[ScoredDocument] =
11   DefaultAccumulator[ScoredDocument]()
12
13 val results: QueryResults = QueryProcessor(rewritten, acc)
```

The only change comes on line 8, where the `Flattener` query rewriter is called to remove interpolated subqueries from the query wherever it can. Note that we explicitly show the insertion of the rewrite step here, but we can easily automate this process more formally by adding a container-type `Rewriters` module that could have rewriters inserted and sequentially run over the query operators before execution. While this is similar to the traversals system in Galago, an important difference is that in Galago, the traversals operate over a tree of `Node` objects that may have no bearing on the actual constructed query - for example, the initial tree after parsing the query cannot be materialized into an executable query. A certain set of traversals *must* operate on the node tree in order to prepare it for execution. In contrast, in Julien the input to, and the output of, every rewriter module here is an executable graph of query operators. Therefore while a certain ordering of rewriters may produce more efficient executions of a particular query, any selection (and any order) of rewriters will produce a valid query operator graph. While this is a significant reduction in complexity, it does have consequences, which we discuss in Section 6.6.

Now that we have determined an appropriate insertion point into the control flow in query processing, all we need is to determine when the rewriter should actively restructure the query. In this case, we must determine the correct context in which this operator should occur (i.e., what is the local query structure) in order to correctly remove the operator.

Assume we are looking at a node N, its parent P, and N's set of children $C = c_1, \ldots, c_j$. We are interested in recognizing when we can safely move c_i to be children of P, thus deleting N from the representation completely. We consider the set of classes that perform the same feature operation (e.g. a `combine` operator and its subclasses all sum over its children during evaluation) to be an equivalence class. We can then use the equivalence class relation to ensure that two operators, which may have different complements of behaviors, are derived from the same base class which actually performs the function of the operator. Based on this definition, the decision criteria we use in the rewriter is simple in this case:

1. Both N and P must be Distributive.

2. N and P must be in an equivalence class.

This logic may seem overly restrictive, however it greatly simplifies the decision-making process during rewriting. Suppose either N or P is *not* Distributive; then if we attempted to perform the flattening operation by moving C under P, either we could not reweight the nodes in C (in the case where N is not Distributive), or we could not set C under P (in the case where P is not Distributive). The second

restriction, that both N and P be the same equivalence class, is to ensure that the same operation will be used over the modified set of children of P.

As an example for the need of the second requirement, Figure 6.5 show a flattening operation when the upper and lower nodes are Distributive (as indicated by the double green outlines), but the operations they implement are different. Although the weight can be distributed over the leaves, the sum operations are deleted in the flattening operation, consequently changing the semantics of the retrieval model.

It is possible to loosen the second restriction to allow subtype equivalences between P and N as long as the core mathematics in the subtype relationship is unchanged. We consider that exploration outside the scope of this work, and instead focus on the effects of single-type query flattening on execution efficiency.

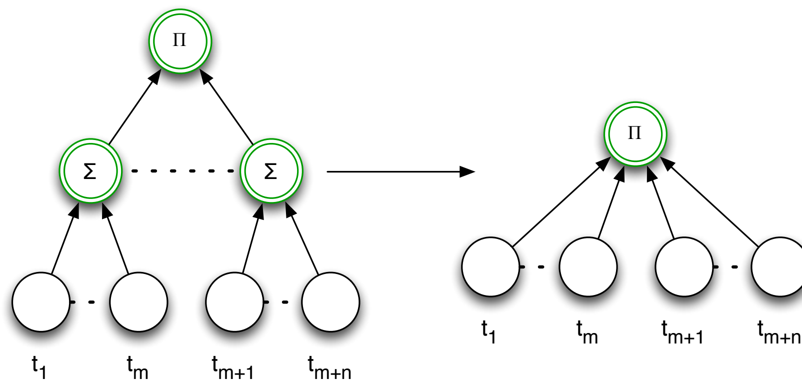

Figure 6.5. Incorrectly flattening the query. The semantics of the query are changed because the lower summation operations are deleted.

As a final implementation note for this section, using the `Distributive` behavior, we can easily define the logic required to determine if a query is eligible for execution using the MAXSCORE or WAND processors: if the root node of the graph

is `Distributive`, we can safely access the children features of the root and directly use them as the scorers in either Maxscore or WAND.

6.5.2 Exposing Alternative Scoring Representations in Julien

We now implement the ASR optimization discussed in Chapter 3. In order to implement ASRs for a particular retrieval model in Julien, we need to add the following functionality:

1. The classes that implement the ASR of the retrieval model.
2. A way to detect when a given query can be converted into its ASR, and
3. A mechanism to convert the original model into its ASR after verifying such an operation will complete successfully.

Implementing item 1 is straightfoward. We can simply implement the new scoring functions as new classes that can be instantiated based on the parameters of the original model. We instead focus our attention on items 2 and 3, as they are more complicated to implement.

In implementing item 2, we could simply require that the ASR implementation be dropped in place of the old implementation, but that requires treatment on a case-by-case basis, and provides no insight towards what the actual requirements are for replacement when we encounter an arbitrary retrieval model.

As an initial solution, we first define the `Bypassable` behavior, which indicates that a feature that exhibits this behavior can generate a set of features that (currently) when summed together produce the same retrieval value as the operator in question. The `Bypassable` behavior exposes the following method:

1. `bypass()` $\rightarrow$ `List[Feature]`

The `bypass` method communicates that the operator can produce a set of alternative operators that will produce the same results during query evaluation, but is arrived at via the new set of operators instead of the original. This mechanism is more general than that required by the ASR examples (PRMS and BM25F) from Chapter 4 - in those instances, we replace the entire set of operators a particular distance from the root. The bypass mechanism can be used locally to replace only parts of the graph, analogous to peephole optimizations performed by a compiler over a local set of machine instructions.

As an example, consider the steps shown in Figure 6.6. The inverted triangles in the graph indicate operators that are in the same equivalence class and all exhibit the `Bypassable` behavior. At step (a), the walk algorithm has marked the first candidate operator. If this had been the only `Bypassable` node, then only that subgraph would be replaced. At step (b), the algorithm has found all the candidate replacement operators. Item 3 from the above implementation list is relatively painless - we can call `bypass` on each of these operators and replace them in situ with their bypassed counterparts. This takes place in step (c) in Figure 6.6, using the graphically shown replacement function below step (b).

Note that while we can implement this sort of operation in the traversal system of Galago, each traversal has limited information about the nodes of the tree.

As before, we can implement the walk algorithm as a query rewrite step that takes place before actual query execution, similar to the modification in Listing 6.6.

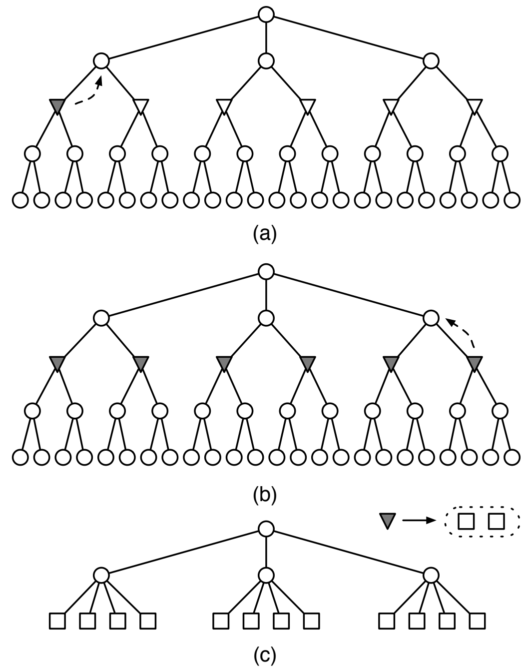

Figure 6.6. A simple walk to look for `Bypassable` operators, which are marked as inverted triangles. The `bypass` function is graphically shown below step (b): a triangle can be replaced by two square operators, which when summed produce the same value as evaluating the entire subtree under the original triangle.

6.5.3 Implementing Delayed Evaluation Julien

We now turn to implementing the last optimization from this thesis, delayed execution, which was presented in Chapter 5. As before, we start off with the general set of requirements we need to meet in order to implement this optimization:

1. We have to create new implementations of the `OrderedWindow` and `UnorderedWindow` operators that can provide estimates of their intersection counts.

2. We must implement a new scorer that takes the estimated count, and uses it to produce high and low estimates of the scores based on the conjunction operators.

3. We need to construct new processors that can take advantage of the two-pass model.

4. We must implement the "completion" routines mentioned in Section 5.1.3.

5. As before, we need a detection and injection mechanism to insert estimated views and features when appropriate, and to choose a processor capable of handling the modified query.

We start with item 1 - new implementations of the conjunction operators. In Chapter 5 we only refer to the estimator functions EstMin and EstMax, however we have to split the implementation of those functions over feature that exposes those two functions with the underlying views of the feature, since the actual work we are trying to elide lies in the view operator (specifically the `OrderedWindow` and `UnorderedWindow` classes). Therefore, we start with new implementations of

those two views, which we call the `ConjunctionEstimator`, which has subclasses of `ODEstimator` and `UWEstimator` to replace `OrderedWindow` and `UnorderedWindow`, respectively. The `ConjunctionEstimator` classes only load count information from the underlying index, so a `count` request is actually the estimated count (currently the min of the underlying counts) of the intersection.

Now that we have views that provide estimated counts (and skip the load of actually performing the intersections), we can create a feature that uses those views to implement EstMin and EstMax from Algorithm 7. In order to indicate that the feature uses estimated counts, we define the `Synthetic` behavior, shown in Listing 6.7.

Listing 6.7. The `Synthetic` trait.

```
1 trait Synthetic {
2   def estMax(id: Int): Double
3   def estMin(id: Int): Double
4 }
```

The `Synthetic` behavior informs downstream code that the operator exposing this behavior creates estimates of scores, and therefore should have scores generated via the `estMin` and `estMax` functions[4].

We have all of the necessary operators available, which allows us to specify queries with estimated components. We now move on to implementing item 3 - creating new processors which can correctly process queries containing estimated components.

---

[4]In reality we can roll these two functions into a single call to cut the number of function calls (and therefore stack overhead) by half, and also use minor dynamic programming to reuse calculated variables between the estimates.

The first part of the implementation is the `AbstractPartialProcessor`. The two important methods of the `AbstractPartialProcessor` are shown in Listing 6.8.

Listing 6.8. The `AbstractPartialProcessor`.

```
1 abstract class AbstractPartialProcessor
2   extends SingleQueryProcessor {
3   def firstPass(): (PriorityQueue, PriorityQueue, Feature)
4   def run(): QueryResult[EstimatedDocument] = {
5     /* code to run the attached completing module */
6   }
7 }
```

The `firstPass` method is abstract, and requires implementation. The implementing classes are shown in the class diagram in Figure 6.7. The two implementing classes, the `DMProcessor` and the `DDProcessor`, correspond to the Sdm-ms-/ and Sdm-msda-/ partial runs from Section 5.2. These two concrete classes complete the necessary implementation for item 3.

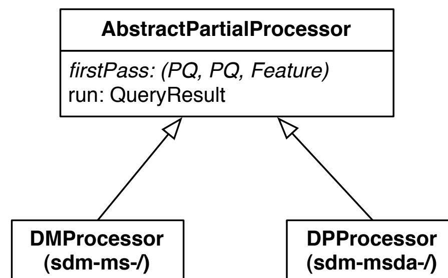

Figure 6.7. A class diagram showing the hierarchy of first-pass processors.

To implement item 4, we actually need two kinds of "completer" routines. The simpler routines (-hi, -lo, and -avg) only require the top and bottom heaps from the

first pass, and the accumulator structure to hold the final results. To accomodate these completers, we define the `SimpleCompleter`, shown in Listing 6.9.

Listing 6.9. The `SimpleCompleter` trait.

```
1 trait SimpleCompleter {
2   def complete(top: PriorityQueue,
3                bottom: PriorityQueue,
4                acc: Accumulator): Unit
5 }
```

The `complete` method returns nothing (as indicated by the `Unit` return type in Scala), as its only function is to carry out the side-effect of updating the `acc` variable. Figure 6.8 shows the class diagram of the implementing `SimpleCompleter` subclasses. The method from Section **??** that the completer implements is shown in parentheses in the diagram.

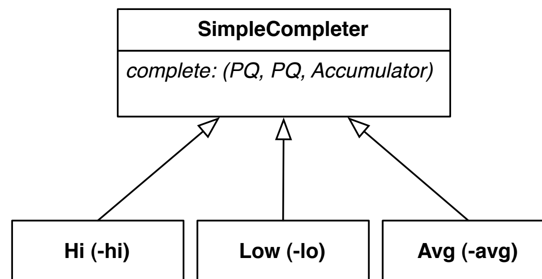

Figure 6.8. A class diagram showing the hierarchy of simple completers.

The remaining completers (-2pass, -ca, -samp, -samp-ca), all perform more processing of the query to improve the estimated results. To enable continued processing, we have to forward the query operator graph to the completer to let it determine what components need completion. We define the `ComplexCompleter` interface in Listing 6.10.

Listing 6.10. The `ComplexCompleter` trait.

```
1 trait ComplexCompleter {
2   def complete(top: PriorityQueue,
3                bottom: PriorityQueue,
4                acc: Accumulator,
5                root: Feature): Unit
6 }
```

Given the interface in Listing 6.10, we can construct our set of completers, as shown in the class diagram in Figure 6.9. Again, the implemented method from Section 5.2 is indicated in the parentheses.

We note that the method signature of `complete` is almost identical to that of the one in `SimpleCompleter`. We could implement both in the same trait/interface, but then for any future completers we would have to implement both methods, which is typically not what an implementor would have in mind, and it would complicate the logic used to call the completion method. By keeping the interfaces separate, we can use simple reflection techniques to determine what kind of completer we are using, and call the appropriate method at runtime.

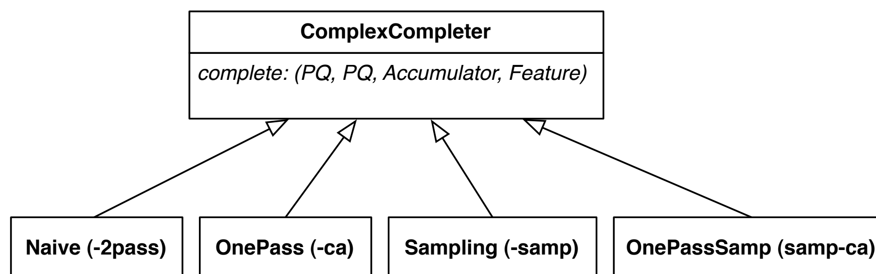

Figure 6.9. A class diagram showing the hierarchy of complex completers.

The final requirement to complete, item 5 (detection and injection), at this point is a simple matter. Since the estimation operators are constructed as part of the executable query graph, we simply can perform a walk over the graph of operators, and look for one or more operators that exhibit the `Synthetic` behavior. If we locate one, we select the combination first-pass processor and completer that we think is most effective, and use that to execute the query.

We have now completed descriptions of implementations using Julien of the optimizations presented in this thesis. Although some of the implementations seemed to involve a large number of classes (i.e. delayed evaluation), many of the classes were written for experimentation, and in a deployment implementation, only the most useful ones would be integrated into the library. The core Julien library is currently available at `https://github.com/mcartright/julien`. The extensions are currently not integrated into the base library, but are constructed as separate packages that use the base library as a dependency. The code for the extensions can be made available upon request.

6.6 The Drawbacks of Julien

No system is without its complications, therefore we would be remiss if we did not examine some of the difficulties one might encounter when using Julien. Some of these difficulties we intend to address in future work, however in the current implementation they are active issues.

Maintenance Requires Learning a New Progamming Language

Julien is written in Scala[5] a hybrid language that draws from both object-oriented and functional programming paradigms. Scala compiles to code that runs on the Java Virtual Machine (JVM), making Java interoperability relatively easy in most cases. This means it is possible to extend the system using Java, and have it make use of compiled core classes, which were written in Scala.

However interoperability typically has limits, and anyone who may need to maintain the Julien core code will need to know Scala. Scala has proven to be a complex language, even for experienced Java programmers. Support and documentation has been growing steadily since the langauge was invented, but the fact remains that Scala requires more time to build proficiency than C++ or Java.

Immediate Query Materialization Forces Greedy Optimizations

Currently queries are immediately materialized when declared in Julien. In Listing 6.3, the declaration of `Dirichlet(/* views */)` actually creates the term-smoothed feature, which can be used to directly generate scores for the provided term view. While this has numerous advantages, it also means that it can be difficult to perform query-level optimizations. For example, consider the query "hubble telescope achievements" from the earlier examples. Suppose we wanted to fork a thread to fully load the posting lists into memory if there was enough space. Notice that if we generalize this problem, it happens to be a variant of the well-known knapsack problem - in fact since we are materializing operators in a particular order

[5]http://scala-lang.org/

(essentially the order imposed by the evaluation requirements of the implementing programming language), it is in fact an *online* knapsack problem - we must decide whether to load the list into memory without having seen all of the other lists we might be able to load yet. The current, and simple, solution would be to use a greedy approach and load until memory is too full to load any more lists. However the current implementation precludes us from trying any solutions that can utilize global (in this case query-level) information to optimize.

No Query Language

Julien was intentionally designed to not have a set query language attached to it, which is a break from most other search engine software available today. Having a set query language allows unfamiliar users to make use of the system without having to learn the internals or write any code to get the system to work. Without it, users are typically relegated to using the implementation language of the system in order to write queries that may require non-default treatment by the system. This will require that users who want to utilize the system for anything beyond simple operation will need to write some small amount of Scala code.

CHAPTER 7

CONCLUSIONS

We have introduced three new optimization techniques for document-at-a-time retrieval systems, as well as a new approach that allows behaviors to be explicitly expressed to the retrieval environment, allowing for on-the-fly optimization situations that would otherwise be difficult to idenitfy.

The first optimization restructures queries containing interpolated subqueries by reducing the depth of a query tree to increase query exposure to a dynamic pruning algorithm. We have empirically shown on two systems that this direct restructuring can reduce average execution time over 80%.

The second optimization builds on insights from the first. In certain cases, retrieval models contain structure that cannot, as yet, be automatically restructured. In order to improve efficiency, we devised a blueprint to reformulate the retrieval model that on average increases execution efficiency by 40%.

The third optimization present here addresses synthetic term dependencies in large queries. By delaying execution of these synthetic components, we can improve execution of these large queries by 20%.

Finally, we introduced Julien, a retrieval framework that focuses on the annotated abilities of operators in a given query, instead of having to hard-code optimizations

into the retrieval pipeline. We have shown that new operators, behaviors, and even processing pipelines are simple to add on top of the core framework. We have implemented the three new optimizations in this framework, and shown their operation in conjunction. This provides an interesting new platform for not only implementing existing state-of-the-art retrieval models and optimizations, but also provides opportunity to consider new aspects of retrieval that previously were difficult to design, and even harder to engineer.

7.1 Relationship to Indri Query Language

The three optimizations presented in this thesis have been shown to work in specific cases, but given their generalized definitions, it is hard to determine the scope of their application - we have a poor notion of the limitations of these optimizations. In this section we inspect the Indri query language (which forms the core of the Galago query language), and shallowly explore which of the optimizations could apply to the operators from this language[1]. Note that the following comparisons must be qualified - a full exploration of the applicability of the optimizations from this thesis to all the operators of the Indri query language is a substantial undertaking, and falls outside the scope of this thesis. However performing this exercise will still provide a better understanding of the scope of these optimizations, even if it is not rigorously defined.

[1]A description of the behavior of each operator can be found in the Appendix.

Operator	Flattening	ASR	Delayed Exec.
#weight/#wand	Y	Y	Y
#combine	Y	Y	Y
#not	N	n/a	n/a
#or	N	Y	Y
#band	Y	Y	Y
#wsum	N	Y	Y
#max	Y	Y	Y
#odN	N	Y	Y
#uwN	N	Y	Y
#syn	Y	Y	Y
#wsyn	Y	Y	Y
#prior	N	N	N
#any	N	Y	Y

Table 7.1. Mapping eligibility of Indri operators for optimization techniques.

We show a matrix between the operators of the Indri query language and the Flattening (Chapter 3), ASR (Chapter 4), and Delayed Execution (Chapter 5) optimizations in Table 7.1. We omit the filtering operators (i.e., `#filrej`, `#filreq`) and comparison operators (e.g., `#greater`, `#dateafter`) as they only serve as a pre-emptive check to decide whether to score a document at all or not. These operators therefore exist outside the normal scoring regime.

When considering whether an optimization is "eligible" to be applied to an operator, we consider the practical benefit of applying the operator. For example, although we could apply delayed execution to the `#combine` operator, such an application confers no clear benefit to processing that operator, so therefore we list the operator as 'not applicable' (marked with an 'n/a'), as opposed to simply ineligible (marked with a 'N').

To gain some insight into the values entered in the matrix, for each optimization we review a case where the optimization would work, where it would not, and where it would be impractical even if it did work (the 'n/a' case).

We begin with the flattening optimization. We already know the optimization works for the `#combine` operator (shown in Chapter 3). We would like to determine what other nodes we can perform with operation on, *without modifying the operation*. In other words, we would like to have a list of operators where it is "legal" to flatten out those operators, but no special steps need to be taken in order to remove a node. They can all be operated on in the same way. Since we have already added `#combine` to this list, the before and after semantics for flattening have already been determined. In order for a node to be eligible for flattening, the node and its parent (if it exists) be in the same equivalence class (as described in Section 6.5.1). Let us consider the `#syn` operator from Table 7.1. We must first construct the candidate case for the operator, say a query such as `#syn(a b #syn(c d e) f g)`. The inner `#syn` is our candidate for removal - it has a parent, and that parent is in the same equivalence class (in this case, it is the exact same operator). If we consider the post-removal query (`#syn(a b c d e f g)`), even through simple inspection we can see that both queries operate the same way - consider the union of all of the children nodes as "hits" for the outer `#syn` operator. Since the before and after queries are semantically equivalent and its input (before) query is an appropriately structured subquery (node and parent are eligible and are in the same equivalence class), we know that we can apply the flattening operation to the `#syn` operator.

Conversely, let us consider why the `#not` node is ineligible for flattening. Practically speaking, `#not` is not an aggregation node, so it may only have one child; flattening this node may not provide much benefit operationally, but one less node in the query graph would equate to one less function call during evaluation. When iterated over millions of documents, removing it could have a noticeable impact.

The purpose of the `#not` node is to invert the belief provided to it. For example, if `#p(a)` is the probabilistic belief that document d is relevant given term a (i.e., $P(d\text{is relevant} \mid a)$), then `#not(#p(a))` is $1 - P(d\text{is relevant} \mid a)$). given this definition, we can determine whether flattening a nesting of this operator is semantically correct. Consider `#not(#not (a))`. Based on our definition, this is equivalent to simply `a`, as the double negation cancels out. Were we to flatten the inner `#not` out, we would get `#not(a)` - which is different semantically from the original query. Despite the input query following the correct form and the node and parent being in the same equivalence class, the before and after query forms are different, meaning we cannot use flattening as is on this operator. It is worth nothing that although *this* verison of flattening is not appropriate, an implementation that recognizes and removes the *pair* of `#not` nodes (i.e., the input query in our example), would properly operate. Although exploring other query configurations available for flattening is beyond the scope of this thesis, this observation indicates that it may make sense to have the operators themselves somehow provide the context necessary to remove themselves from the query graph.

We now inspect the case of ASR eligibility. We consider an operator to be eligible if it is reasonably conceivable that a rewritten form of the operator (or the subtree

under it) could be written such that either 1) it exposes more of the subtree for dynamic pruning mechanisms such as Maxscore, or 2) it produces a smaller subtree, therefore simply requiring less computation to compute the subtree. As before, since we have applied the optimization to the PRMS and BM25F retrieval models, we have defined the parameters for the optimization to be applied: a node must report that it can be replaced by a different set of semantically equivalent operators (in the case of Julien, this is by having the operator exhibit the `Bypassable` behavior). Exhaustively examining each case is impractical, since the effectiveness of the optimization partially rests on the way in which the mathematics of the operator are manipulated. Instead we consider any aggregation node (i.e., a operator that accept one or more arguments) to be eligible for ASR rewriting. We consider ASR rewriting impractical for the non-aggregation operators; for this exercise we assume a single operator will execute faster than a semantically equivalent set of them. Hence, all non-aggregation nodes are marked 'n/a', since we *could* rewrite them, but it seems unrealistic to do so.

A `#prior` operator requires special mention. This operator is considered ineligible for ASR rewriting due to its implementation - priors are stored a list of stored beliefs, one per field per document. We assume that recovering a single number from the index is an atomic operation, and so for the scope of this exercise, nothing would be faster than performing that action, which is the action taken by the `#prior` operator.

Finally, we consider eligibility for Delayed Execution. If an operator is eligible for Delayed Execution, that means it would be possible to write estimator versions of the operator, and then correct for the estimations later, similar to the steps performed

in Chapter 5. As shown in that chapter, we know both `#odN` and `#uwN` are eligible for Delayed Execution. Similar to the ASR rewrite, we speculate that all aggregation operators are eligible for Delayed Execution, although the practicality of applying the optimization would have to be examined - in Chapter 5 we performed this analysis for `#odN` and `#uwN` - but the other operators would require similar inspection. However, if we are optimistic and assume that every aggregation operator could be accurately estimated, and not actually evaluated, then those operators form out set of eligible nodes. As before, the `#prior` and `#not` operators are not eligible. In the case of the former, the operation is already as cheap as it can get - estimating the value would, for the most part, cost as much as actually obtaining the value. In the case of the latter, since the operator is a simple inversion transform on its argument, in this case its complexity is solely dependent on its argument (otherwise it is an extremely cheap constant-time operation). In this exercise, it is not practical to bother delaying the execution of the `#not` operator, but an interesting future approach would be to transitively carry the complexity of the operators up through the query graph, in which case delayed execution the `#not` operator may make sense when its argument is expensive.

7.2 Future Work

We discuss both new opportunities for optimization as well as exploring new retrieval models that may have been difficult to realize using prior systems.

Optimization Opportunity: Determining the Next Candidate

Consider in a daat processing system, where all of the index-level pointers are lined up to a candidate document as best as possible, and all scoring functions are evaluated to fully score the candidate. If the retrieval model is defined by a set of disjunctive scoring functions (e.g., a simple keyword model), then for any given candidate, the next candidate can easily be determined by moving each of the index pointers past the current candidate, and selecting the minimum next candidate from the index pointers. If the retrieval system supports conjunction operations such as scoring on phrases or windows of text, then determining the next candidate is no longer as simple, since, by the definition of the next candidate, the conjunction operations should move the underlying index pointers forward until they all land on the same document. This can be achieved by a simple loop of choosing the max of the involved pointers and attempting to align to that value. If no match is found, all pointers are moved past that candidate, and a new max is found. Shared index pointers complicates the issue further; while a pointer in a conjunction should be moved aggressively (find max), if that pointer is also used in a disjunction, it must be moved conservatively (find min), therefore requiring two different behaviors from the same pointer.

We would like to explore the different possibilities for optimally traversing the index under such conditions, specifically by using different versions of the same conjunctive operators (each with different available behaviors) for the cases of only passive, only aggressive, or mixed-mode movement.

Optimization Opportunity: Asynchronous Query Evaluation

The abstraction of views away from the component that provides index-level information begs an interesting question: instead of performing query evaluation serially, either by keeping all pointers in lock-step, or traversing entire posting lists one at a time, can we simply try to have the system read the data as fast as possible, and use a publish/subscribe model to generate the final ranked list? This approach uses the same intuition as the dataflow (or more recently the reactive programming (Demetrescu, Finocchi, & Ribichini, 2011)) model, where the computation occurs when there is data ready to be processed. The elimination of having to check positioning in the posting lists may provide a significant boost in performance, however we may trade that off with an increase in traffic when updating the various data structures needed to accumulate partial results. We would like to explore this approach to query evaluation to determine if any potential lies in letting the index pointers drive the evaluation instead of using external logic to drag the pointers forward as needed.

Optimization Application: Integrated Model Parameterization

Most parameterization procedures occur outside the knowledge of the retrieval system. The system is typically called repeatedly, with different parts of a query receiving slightly different weights, until a general optimum is found over a set of training queries. If the parameterization could be generalized to inform the system how to search a parameter space to optimize a particular query or set of queries, the system could heavily optimize processing by caching certain results or query subtrees to save on needlessly rerunning the same computation. We have already

extended Julien to automatically tune the BM25F model to optimum weighting for a selected set of training queries using a "Tunable" behavior. We plan to generalize this behavior to support parameterization of any given retrieval model that can be generated using a Tunable factory mechanism.

System Application: Query-Sensitive Retrievable-Level Dependencies

Retrievable-level dependencies have been useful tools in retrieval so far, but to date they have been used in isolated ways, such as in single-iteration cycles (e.g., PRF) or offline procedures such as constructing link graphs between the retrievables. The idea of jointly exploring retrievable-level dependencies conditioned on the query context has to date not been deeply explored. It may be that such dependencies are ultimately not informative, however it may also be that even expressing such dependencies efficiently has been difficult at best. Recent research suggests that exploiting such dependencies may allow for significant improvements in retrieval performance (Dietz & Dalton, 2013; Maxwell & Croft, 2013). We would like to extend Julien to support such interactions between the query and the set of retrievables. The extension would enable research into models that better capture the interplay of information between the query and the documents the system presents to the user. For example, examining results as they are generated would make it a much simpler matter to diversify results during the initial retrieval, instead of waiting for the initial round to end, and performing topic diversification as a rerank step.

System Application: The Proteus Project

The Proteus project is the result of a collaborative effort between the University of Massachusetts, the Internet Archive, and the Perseus Project from Tufts University, and more recently Northeastern University. Using digitally scanned books provided by the IA, UMass is working towards a system that allows a combination of searching and browsing over books and associated extracted data (Cartright, Can, et al., 2012; Cartright, Dalton, & Allan, 2012).

The underlying system was originally implemented using Galago. An index was created for each retrievable type, and when processing a query, each of these indexes was queried independently and in parallel. This approach works well for a context-less query - lacking a context, we may assume independence of the retrieval types and send the query to be processed in parallel. However, this paradigm breaks down when we would like to perform retrieval using some form of feedback from the user. For example, it is currently possible to issue a query such as "beer brewing" and retrieve results on books, locations, and people associated with the query. However, if a user wants to perform the same query, but with the feedback that the person "William Sealy Gosset" and the location "Dublin" have been marked as relevant, we do not have a model that can effectively leverage this information. We would like to use Julien to support retrieval *using* multiple retrieval types as feedback information, as well as retrieving multiple retrieval types. Ultimately, we would like to incorporate ideas from retrievable-level dependencies to return entire sets of retrieved items as a single composite item, as opposed to returning each retrievable in isolation.

APPENDIX

OPERATORS FROM THE INDRI QUERY LANGUAGE

As of August 29, 2013, the definitive reference for the Indri query language can be found in two locations:

1. `http://sourceforge.net/p/lemur/wiki/The\%20Indri\%20Query\%20Language/`
2. `http://www.lemurproject.org/lemur/IndriQueryLanguage.php`

The information listed here is a selected reprinting of the reference found there at this time, in order to provide a self-contained reference for this thesis. Only the operators referenced in this thesis are described here. Please refer to the online reference for details concerning the other operators. In the listing below, the function `b` refers to any belief operator defined in the Indri query language, e.g., the `#combine` operator.

List of Operators

Operator Name: document prior

Example: `#prior(RECENT)`

Behavior: If the document contains a value for the prior `RECENT`, this belief value will be factored into the weighting of the document.

Operator Name: weight, weighted and

Example: `#weight(1.0 dog 0.5 train)` **-or-** `#wand(1.0 dog 0.5 train)`

Behavior: $0.67 \log(\texttt{b(dog)}) + 0.33 \log(\texttt{b(train)})$

Operator Name: combine

Example: `#combine(dog train)`

Behavior: $0.5 \log(\texttt{b(dog)}) + 0.5 \log(\texttt{b(train)})$

Operator Name: not

Example: `#not(dog)`

Behavior: $\log(1 - \texttt{b(dog)})$

Operator Name: or

Example: `#or(dog cat)`

Behavior: $\log(1 - (1 - \texttt{b(dog)}) * (1 - \texttt{b(cat)}))$

Operator Name: boolean and

Example: `#band(cat dog)`

Behavior: Produces a single extent of 1 if both `cat` and `dog` are present. Produces no extents otherwise.

Operator Name: weighted sum

Example: `#wsum(1.0 dog 0.5 dog.(title))`

Behavior: $\log(0.67\texttt{b(dog)} + 0.33\texttt{b(dog.(title))})$

Operator Name: max

Example: `#max(dog train)`

Behavior: Returns maximum of `b(dog)` and `b(train)`.

Operator Name: ordered window

Example: `#od‘‘n’’(blue car)` **-or-** `#‘‘n’’(blue car)`

Behavior: `blue` appears “n” words or less before `car`.

Operator Name: unordered window

Example: `#uw‘‘n’’(blue car)`

Behavior: `blue` within “n” words of `car`.

Operator Name: synonym list

Example: `#syn(car automobile)`

Behavior: Occurrences of `car` or `automobile`.

Operator Name: weighted synonym

Example: `#wsyn(1.0 car 0.5 automobile)`

Behavior: Like synonym, but only counts occurrences of `automobile` as 0.5 of an occurrence.

Operator Name: any

Example: `#any:person`

Behavior: All occurrences of the `person` field.

REFERENCES

Abdul-jaleel, N., Allan, J., Croft, W. B., Diaz, O., Larkey, L., Li, X., ... Wade, C. (2004). UMass at TREC 2004: Notebook. In *Trec 2004* (pp. 657–670).

Allan, J. (2002). Introduction to Topic Detection and Tracking. In J. Allan & W. B. Croft (Eds.), *Topic detection and tracking* (Vol. 12, p. 1-16). Springer US.

Allen, R., & Kennedy, K. (2001). *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann.

Anh, V. N., de Kretser, O., & Moffat, A. (2001). Vector-space ranking with effective early termination. In *Proceedings of the 24th annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 35–42). New York, NY, USA: ACM.

Anh, V. N., & Moffat, A. (2006). Pruned Query Evaluation using Pre-Computed Impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)* (pp. 372–379). New York, NY, USA: ACM.

Apache Software Foundation, T. (2012, March). *Lucene Query Language* [website]. Retrieved from `http://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/queryparsersyntax.html`

Arnt, A., Zilberstein, S., Allan, J., & Mouaddib, A.-I. (2004). Dynamic Composition of Information Retrieval Techniques. *Journal of Intelligent Information Systems*, *23*, 67-97.

Baeza-Yates, R., Castillo, C., Junqueira, F., Plachouras, V., & Silvestri, F. (2007). Challenges on distributed web retrieval. In *Data engineering, 2007. icde 2007. ieee 23rd international conference on* (pp. 6–20).

Bendersky, M. (2012). personal communication. (February 23, 2012)

Bendersky, M., & Croft, W. B. (2008). Discovering key concepts in verbose queries. In *Proceedings of the 31st SIGIR* (pp. 491–498). New York, NY, USA: ACM.

Bendersky, M., Metzler, D., & Croft, W. B. (2011). Parameterized Concept Weighting in Verbose Queries. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 605–614). New York, NY, USA: ACM.

Blandford, D., & Blelloch, G. (2002). Index Compression through Document Reordering. In *Proceedings of the data compression conference* (pp. 342–). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dl.acm.org/citation.cfm?id=882455.875020`

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on information and Knowledge Management* (pp. 426–434). New York, NY, USA: ACM.

Brown, E. W. (1995). Fast Evaluation of Structured Queries for Information Retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)* (pp. 30–38). New York, NY, USA: ACM.

Buckley, C., & Lewit, A. F. (1985). Optimization of Inverted Vector Searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 97–110). New York, NY, USA: ACM.

Büttcher, S., & Clarke, C. L. A. (2006). A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 15th acm international conference on information and Knowledge Management* (pp. 182–189). New York, NY, USA: ACM.

Callan, J., Croft, W. B., & Harding, S. M. (1992). The INQUERY Retrieval System. In *In proceedings of the third international conference on database and expert systems applications* (pp. 78–83). Springer-Verlag.

Cao, G., Nie, J.-Y., Gao, J., & Robertson, S. (2008). Selecting good expansion terms for pseudo-relevance feedback. In *Proceedings of the 31st annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 243–250). New York, NY, USA: ACM.

Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y. S., & Soffer, A. (2001). Static index pruning for Information Retrieval systems. In *Proceedings of the 24th annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 43–50). New York, NY, USA: ACM.

Cartright, M.-A., & Allan, J. (2011). Efficiency Optimizations for Interpolating Subqueries. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (pp. 297–306). New York, NY, USA: ACM.

Cartright, M.-A., Can, E. F., Dabney, W., Dalton, J., Giorda, L., Krstovski, K., ... others (2012). A Framework for Manipulating and Searching Multiple Retrieval Types. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 1001–1001).

Cartright, M.-A., Dalton, J., & Allan, J. (2012). Search and Exploration of Scanned Books. In *Proceedings of the Fifth ACM Workshop on Research Advances in Large Digital Book Repositories and Complementary Media* (pp. 9–10).

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*(2), 4.

Chaudhuri, S. (1998). An Overview of Query Optimization in Relational Systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (pp. 34–43).

Clarke, C. L. A., Cormack, G. V., & Burkowski, F. J. (1995). An algebra for structured text search and a framework for its implementation. *The Computer Journal*, *38*, 43–56.

Cohen, W. W., & Hirsh, H. (1998). Joins that Generalize: Text Classification Using WHIRL. In *In Proc. of the Fourth Int'l Conference on Knowledge Discovery and Data Mining* (pp. 169–173).

Craswell, N., & Szummer, M. (2007). Random Walks on the Click Graph. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 239–246). New York, NY, USA: ACM.

Croft, W. B., Metzler, D., & Strohman, T. (2010). *Search Engines: Information Retrieval in Practice*. Addison-Wesley Reading.

Culpepper, J. S., Petri, M., & Scholer, F. (2012). Efficient in-memory top-k document retrieval. In *Proceedings of the 35th international acm SIGIR conference on research and development in Information Retrieval* (pp. 225–234). New York, NY, USA: ACM.

Dalgarno, B., & Lee, M. J. (2010). What are the learning affordances of 3-d virtual environments? *British Journal of Educational Technology*, *41*(1), 10–32.

Dalvi, N., & Suciu, D. (2007). Efficient Query Evaluation on Probabilistic Databases. *The VLDB Journal*, *16*, 523-544. (10.1007/s00778-006-0004-3)

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Vol. 7, pp. 205–220).

de Kunder, M. (2012, March). *The Size of the World Wide Web (The Internet)* [website]. Retrieved from `http://worldwidewebsize.com`

Demetrescu, C., Finocchi, I., & Ribichini, A. (2011). Reactive Imperative Programming with Dataflow Constraints. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (pp. 407–426). New York, NY, USA: ACM.

Dennis, S. F. (1964). The Construction of a Thesaurus Automatically from a Sample of Text. In *Symposium on statistical methods fro mechanized documentation.*

Dietz, L., & Dalton, J. (2013). Constructing Query-Specific Knowledge Bases. In *Automated Knowledge Base Construction (AKBC) 2013.*

Efron, B., & Tibshirani, R. (1993). *An Introduction to the Bootstrap.* New York : Chapman & Hall.

Entlich, R., Olsen, J., Garson, L., Lesk, M., Normore, L., & Weibel, S. (1997, April). Making a Digital Library: The Contents of the CORE Project. *ACM Transactions on Information Systems*, *15*, 103–123.

Fagg, A. H., & Arbib, M. A. (1998). Modeling parietal–premotor interactions in primate control of grasping. *Neural Networks*, *11*(7), 1277–1303.

Fagin, R. (1996). Combining fuzzy information from multiple systems (extended abstract). In *Proceedings of the fifteenth acm sigact-sigmod-sigart symposium on principles of database systems* (pp. 216–226). New York, NY, USA: ACM.

Fisher, D. (2013). personal communication. (July 9, 2013)

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

Fox, E. A., & Sornil, O. (2003). Digital Libraries. In *Encyclopedia of Computer Science* (pp. 576–581). Chichester, UK: John Wiley and Sons Ltd.

Fuhr, N., Gövert, N., Kazai, G., & Lalmas, M. (2002). INEX: INitiative for the Evaluation of XML retrieval. *Proceedings of the SIGIR 2002 Workshop on XML and Information Retrieval*, 1–9.

Gaver, W. W. (1991). Technology affordances. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 79–84).

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. In *Acm sigops operating systems review* (Vol. 37, pp. 29–43).

Gibson, J. J. (1977). The Theory of Affordances. *Perceiving, Acting, and Knowing: Toward an Ecological Psychology*.

Gil-Costa, V., Lobos, J., Inostrosa-Psijas, A., & Marin, M. (2012). Capacity planning for vertical search engines: An approach based on coloured petri nets. In *Application and theory of petri nets* (pp. 288–307). Springer.

Harman, D. (1993). Overview of TREC-1. In *Proceedings of the Workshop on Human Language Technology* (pp. 61–65). Stroudsburg, PA, USA: Association for Computational Linguistics.

He, J., Zeng, J., & Suel, T. (2010). Improved Index Compression Techniques for Versioned Document Collections. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management* (pp. 1239–1248). New York, NY, USA: ACM.

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.

Huston, S., Moffat, A., & Croft, W. B. (2011). Efficient Indexing of Repeated n-Grams [IR]. In *Fourth acm international conference on web search and data mining.*

Ingersoll, G. (2012). personal communication. (August 15, 2012)

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, *41*(3), 59–72.

Jonassen, S. (2012). Scalable Search Platform: Improving Pipelined Query Processing for Distributed Full-Text Retrieval. In *Proceedings of the 21st International World Wide Web Conference* (pp. 145–150).

Kamvar, M., & Baluja, S. (2007, aug.). Deciphering Trends in Mobile Search. *Computer*, *40*(8), 58 -62.

Kim, J., & Croft, W. B. (2009). Retrieval experiments using pseudo-desktop collections. In *Proceedings of the 18th acm conference on information and Knowledge Management* (pp. 1297–1306). New York, NY, USA: ACM.

Kim, J., & Croft, W. B. (2010). Ranking using multiple document types in desktop search. In *Proceedings of the 33rd international acm SIGIR conference on research and development in Information Retrieval* (pp. 50–57). New York, NY, USA: ACM.

Kim, J., Xue, X., & Croft, W. B. (2009). A probabilistic retrieval model for semistructured data. In *Proceedings of the 31th european conference on ir research on advances in Information Retrieval* (pp. 228–239). Berlin, Heidelberg: Springer-Verlag.

Koller, D., & Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

Lavrenko, V., & Croft, W. B. (2001). Relevance based language models. In *Proceedings of the 24th SIGIR* (pp. 120–127). New York, NY, USA: ACM.

Lin, J., Metzler, D., Elsayed, T., & Wang, L. (2010). Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. In *Trec 2010.*

Lu, Y., Peng, F., Wei, X., & Dumoulin, B. (2010). Personalize Web Search Results with User's Location. In *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 763–764). New York, NY, USA: ACM.

Macdonald, C., Ounis, I., & Tonellotto, N. (2011, December). Upper-Bound Approximations for Dynamic Pruning. *ACM Transactions on Information Systems*, *29*, 17:1–17:28.

Macdonald, C., Plachouras, V., He, B., & Ounis, I. (2004). University of Glasgow at TREC 2005: Experiments in Terabyte and Enterprise Tracks with Terrier. In *Proceedings of TREC 2005.*

Maisonnasse, L., Gaussier, E., & Chevallet, J.-P. (2007). Revisiting the Dependence Language Model for Information Retrieval. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 695–696). New York, NY, USA: ACM.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval* (Vol. 1). Cambridge University Press Cambridge.

Maxwell, K. T., & Croft, W. B. (2013). Compact Query Term Selection using Topically Related Text. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 583–592). New York, NY, USA: ACM.

McGrenere, J., & Ho, W. (2000). Affordances: Clarifying and evolving a concept. In *Graphics interface* (Vol. 2000, pp. 179–186).

Metzler, D., & Croft, W. B. (2005). A markov random field model for term dependencies. In *Proceedings of the 28th annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 472–479). New York, NY, USA: ACM.

Metzler, D., & Croft, W. B. (2007). Latent Concept Expansion using Markov Random Fields. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 311–318).

Moffat, A., Webber, W., Zobel, J., & Baeza-Yates, R. (2007a). A Pipelined Architecture for Distributed Text Query Evaluation. *Information Retrieval*, *10*(3), 205–231.

Moffat, A., Webber, W., Zobel, J., & Baeza-Yates, R. (2007b, June). A Pipelined Architecture for Distributed Text Query Evaluation. *INformation REtrieval*, *10*(3), 205–231.

Norman, D. A. (2002). *The Design of Everyday Things*. Basic Books (AZ).

Ogilvie, P., & Callan, J. (2003). Combining Document Representations for Known-Item Search. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval* (pp. 143–150). New York, NY, USA: ACM.

Ogilvie, P., & Callan, J. (2005). Hierarchical language models for xml component retrieval. In *Advances in xml Information Retrieval* (pp. 224–237). Springer.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999, November). *The PageRank Citation Ranking: Bringing Order to the Web.* (Technical Report No. 1999-66). Stanford InfoLab. (Previous number = SIDL-WP-1999-0120)

Park, J. H., Croft, W. B., & Smith, D. A. (2011). A Quasi-Synchronous Dependence Model for Information Retrieval. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (pp. 17–26). New York, NY, USA: ACM.

Ponte, J. M., & Croft, W. B. (1998). A language modeling approach to Information Retrieval. In *Proceedings of the 21st SIGIR* (pp. 275–281). New York, NY, USA: ACM.

Robertson, S., Zaragoza, H., & Taylor, M. (2004). Simple BM25 Extension to Multiple Weighted Fields. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management* (pp. 42–49). New York, NY, USA: ACM.

Robertson, S. E., & Walker, S. (1994). Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th SIGIR* (pp. 232–241). New York, NY, USA: Springer-Verlag New York, Inc.

Rocchio, J. (1971). Relevance feedback in Information Retrieval. In G. Salton (Ed.), *The smart retrieval system: Experiments in automatic document processing* (pp. 313–323). Englewood Cliffs, NJ: Prentice-Hall.

Salton, G. (1971). The SMARt Retrieval SystemExperiments in Automatic Document Processing.

Salton, G., Wong, A., & Yang, C. S. (1975, November). A vector space model for automatic indexing. *Communications of the ACM*, *18*, 613–620.

Sanderson, M., & Croft, W. B. (2012). The History of Information Retrieval Research. *Proceedings of the IEEE*, *100*(13), 1444–1451.

Schenkel, R., Broschart, A., Hwang, S., Theobald, M., & Weikum, G. (2007). Efficient text proximity search. In *Proceedings of the 14th international conference on string processing and Information Retrieval* (pp. 287–299). Berlin, Heidelberg: Springer-Verlag.

Schurman, E., & Brutlag, J. (2009). The user and business impact of server delays, additional bytes, and http chunking in web search. In *Presentation at the oreilly velocity web performance and operations conference.*

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 acm sigmod International Conference on Management of Data* (pp. 23–34). New York, NY, USA: ACM.

Sen, S., Sherrick, G., Ruiken, D., & Grupen, R. A. (2011). Hierarchical skills and skill-based representation. In *Lifelong learning.*

Silvestri, F. (2007). Sorting Out the Document Identifier Assignment Problem. In G. Amati, C. Carpineto, & G. Romano (Eds.), *Advances in Information Retrieval* (Vol. 4425, p. 101-112). Springer Berlin / Heidelberg.

Spink, A., Wolfram, D., Jansen, M. B. J., & Saracevic, T. (2001). Searching the Web: The Public and Their Queries. *Journal of the American Society for Information Science and Technology*, *52*(3), 226–234.

StatCounter. (2011, July). *Top 5 Search Engines from Feb to July 2011* [website]. Retrieved from `http://gs.statcounter.com/#search_engine-ww-monthly-201102-201107-bar`

Stoytchev, A. (2005a). Behavior-grounded representation of tool affordances. In *Robotics and automation, 2005. icra 2005. proceedings of the 2005 ieee international conference on* (pp. 3060–3065).

Stoytchev, A. (2005b). Toward learning the binding affordances of objects: A behavior-grounded approach. In *Proceedings of aaai symposium on developmental robotics* (pp. 17–22).

Strohman, T. (2007). *Efficient processing of complex features for Information Retrieval.* Ph.D. dissertation, University of Massachusetts Amherst.

Strohman, T., & Croft, W. B. (2007). Efficient document retrieval in main memory. In *Proceedings of the 30th annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 175–182). New York, NY, USA: ACM.

Strohman, T., Metzler, D., Turtle, H., & Croft, W. B. (2005). Indri: A Language Model-Based Search Engine for Complex Queries. In *Proceedings of the International Conference on Intelligent Analysis* (Vol. 2, pp. 2–6).

Strohman, T., Turtle, H., & Croft, W. B. (2005). Optimization strategies for complex queries. In *Proceedings of the 28th annual international acm SIGIR conference on research and development in Information Retrieval* (pp. 219–225). New York, NY, USA: ACM.

Svore, K. M., Kanani, P. H., & Khan, N. (2010). How good is a span of terms?: exploiting proximity to improve web retrieval. In *Proceeding of the 33rd international acm SIGIR conference on research and development in Information Retrieval* (pp. 154–161). New York, NY, USA: ACM.

Swanson, D. R. (1960). Searching Natural Language Text by Computer. *Science*, *132*(3434), 1099-1104.

Teevan, J., Ramage, D., & Morris, M. R. (2011). #twittersearch: A Comparison of Microblog Search and Web Search. In *Proceedings of the Fourth ACM International Conference on web Search and Data Mining* (pp. 35–44). New York, NY, USA: ACM.

Tonellotto, N., Macdonald, C., & Ounis, I. (2011). Effect of different docid orderings on dynamic pruning retrieval strategies. In *Proceedings of the 34th international acm SIGIR conference on research and development in Information Retrieval* (pp. 1179–1180). New York, NY, USA: ACM.

Tonellotto, N., Macdonald, C., & Ounis, I. (2013). Efficient and effective retrieval using selective pruning. In *Proceedings of the sixth acm international conference on web search and data mining* (pp. 63–72).

Trotman, A. (2012). personal communication. (August 15, 2012)

Turtle, H., & Croft, W. B. (1990). Inference networks for document retrieval. In *Proceedings of the 13th SIGIR* (pp. 1–24). New York, NY, USA: ACM.

Turtle, H., & Croft, W. B. (1991, July). Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, *9*, 187–222.

Turtle, H., & Flood, J. (1995, November). Query evaluation: strategies and optimizations. *Information Processing & Management*, *31*, 831–850.

Turtle, H., Morton, G. J., & Larntz, F. K. (1996). *System of document representation retrieval by successive iterated probability sampling* (Tech. Rep.). United States Patent Office. (US Patent 5,488,725)

University of Glasgow, S. o. C. (2011). *The Terrier IR Platform.* Retrieved from `http://terrier.org`

Wang, L., Lin, J., & Metzler, D. (2010). Learning to Efficiently Rank. In *Proceedings of the 33rd international acm SIGIR conference on research and development in Information Retrieval* (pp. 138–145). New York, NY, USA: ACM.

Wang, L., Lin, J., & Metzler, D. (2011). A Cascade Ranking Model for Efficient Ranked Retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 105–114). New York, NY, USA: ACM.

Wang, L., Metzler, D., & Lin, J. (2010). Ranking under temporal constraints. In *Proceedings of the 19th acm international conference on information and Knowledge Management* (pp. 79–88). New York, NY, USA: ACM.

Wong, E., & Youssefi, K. (1976, September). Decomposition - A Strategy for Query Processing. *ACM Transactions on Database Systems*, *1*(3), 223–241.

Xu, J., & Li, H. (2007). Adarank: A Boosting Algorithm for Information Retrieval. In *Proceedings of the 30th Annual International acm SIGIR Conference on Research and Development in Information Retrieval* (pp. 391–398). New York, NY, USA: ACM.

Xue, X., Huston, S., & Croft, W. B. (2010). Improving verbose queries using subset distribution. In *Proceedings of the 19th acm international conference on information and Knowledge Management* (pp. 1059–1068). New York, NY, USA: ACM.

Yan, H., Ding, S., & Suel, T. (2009a). Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on world wide web* (pp. 401–410). New York, NY, USA: ACM.

Yan, H., Ding, S., & Suel, T. (2009b). Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on world wide web* (pp. 401–410). New York, NY, USA: ACM.

Yan, H., Shi, S., Zhang, F., Suel, T., & Wen, J.-R. (2010). Efficient Term Proximity Search with Term-Pair Indexes. In I. M. Sheepish (Ed.), *Proceedings of the nineteenth international conference on information and Knowledge Management* (pp. 39–45). Toronto, Ontario, Canada: ACM.

Yi, X., Allan, J., & Croft, W. B. (2007). Matching resumes and jobs based on relevance models. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 809–810). New York, NY, USA: ACM.

Yi, X., Raghavan, H., & Leggetter, C. (2009). Discovering users' specific geo intention in web search. In *Proceedings of the 18th international conference on world wide web* (pp. 481–490). New York, NY, USA: ACM.

Zaragoza, H., Craswell, N., Taylor, M., Saria, S., & Robertson, S. (2004). Microsoft Cambridge at TREC-13: Web and Hard tracks. In *Proceedings of TREC-2004.*

Zhu, M., Shi, S., Li, M., & Wen, J.-R. (2007). Effective top-k computation in retrieving structured documents with term-proximity support. In *Proceedings of the sixteenth acm conference on conference on information and Knowledge Management* (pp. 771–780). New York, NY, USA: ACM.

Zobel, J., Williams, H., Scholer, F., Yiannis, J., & Hein, S. (2004). The Zettair Search Engine. *Search Engine Group, RMIT University, Melbourne, Australia*.

Zukowski, M., Heman, S., Nes, N., & Boncz, P. (2006). Super-scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering* (pp. 59–59).