Simon Jonassen

# Efficient Query Processing in Distributed Search Engines

Thesis for the degree of Philosophiae Doctor

Trondheim, January 2013

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

**NTNU – Trondheim**
Norwegian University of
Science and Technology

**NTNU**
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

© Simon Jonassen

ISBN 978-82-471-4134-2 (printed ver.)
ISBN 978-82-471-4135-9 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2013:22

Printed by NTNU-trykk

*To Inna, Elmira and Anne Margrethe.*

# Abstract

Web search engines have to deal with a rapidly increasing amount of information, high query loads and tight performance constraints. The success of a search engine depends on the speed with which it answers queries (efficiency) and the quality of its answers (effectiveness). These two metrics have a large impact on the operational costs of the search engine and the overall user satisfaction, which determine the revenue of the search engine. In this context, any improvement in query processing efficiency can reduce the operational costs and improve user satisfaction, hence improve the overall benefit.

In this thesis, we elaborate on query processing efficiency, address several problems within partitioned query processing, pruning and caching and propose several novel techniques:

First, we look at term-wise partitioned indexes and address the main limitations of the state-of-the-art query processing methods. Our first approach combines the advantage of pipelined and traditional (non-pipelined) query processing. This approach assumes one disk access per posting list and traditional term-at-a-time processing. For the second approach, we follow an alternative direction and look at document-at-a-time processing of sub-queries and skipping. Subsequently, we present several skipping extensions to pipelined query processing, which as we show can improve the query processing performance and/or the quality of results. Then, we extend one of these methods with intra-query parallelism, which as we show can improve the performance at low query loads.

Second, we look at skipping and pruning optimizations designed for a monolithic index. We present an efficient self-skipping inverted index designed for modern index compression methods and several query processing optimizations. We show that these optimizations can provide a significant speed-up compared to a full (non-pruned) evaluation and reduce the performance gap between disjunctive (OR) and conjunctive (AND) queries. We also propose a linear programming optimization that can further improve the I/O, decompression and computation efficiency of Max-Score.

Third, we elaborate on caching in Web search engines in two independent contributions. First, we present an analytical model that finds the optimal split in a static memory-based two-level cache. Second, we present several strategies for selecting, ordering and scheduling prefetch queries and demonstrate that these can improve the efficiency and effectiveness of Web search engines.

We carefully evaluate our ideas either using a real implementation or by simulation using real-world text collections and query logs. Most of the proposed techniques are found to improve the state-of-the-art in the conducted empirical studies. However, the implications and applicability of these techniques in practice need further evaluation in real-life settings.

# Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree of philosophiae doctor. The work has been performed at the Department of Computer and Information Science, NTNU, Trondheim and Yahoo! Research, Barcelona. The doctoral program has been supervised by Professor Svein Erik Bratsberg, Dr. Øystein Torbjørnsen (Microsoft Development Center Norway) and Associate Professor Magnus Lie Hetland. At Yahoo, the work has been mentored by Professor Ricardo Baeza-Yates and Dr. B. Barla Cambazoglu. During the process, the candidate has been supported by the Information Access Disruptions (iAd) project and funded by the Research Council of Norway and NTNU.

# Acknowledgments

First and foremost, I thank my supervisors, Prof. Svein Erik Bratsberg and Dr. Øystein Torbjørnsen, for their support and encouragement. Our meetings were an important source of knowledge and experience, which helped me to discover, structure and clarify the ideas presented in this thesis.

Further, I thank my colleagues at the Department of Computer and Information Science for a great environment, fruitful discussions, inspiration and advice. Dr. Magnus Lie Hetland, Dr. Truls A. Bjørklund and Dr. Nils Grimsmo taught me algorithms and critical thinking back in 2004 and later motivated me to start on a PhD. I thank Ola Natvig for SolBrille, bit-hacks and coffee breaks during my first year as a PhD student. The feedback from Dr. João B. Rocha-Junior and Robert Neumayer improved many of my papers and taught me research and scientific writing. The company of Dr. Marek Ciglan, Naimdjon Takhirov, Massimiliano Ruocco and Dr. Nattiya Kanhabua turned the conference travels into exciting adventures. Special thanks to the HPC Group for allowing me to use their cluster.

I extend my gratitude to Prof. Ricardo Baeza-Yates for giving me an amazing opportunity to stay at the Yahoo! Research lab – a fun and creative environment, which became my second home during the time spent in Barcelona. I am deeply grateful to Dr. B. Barla Cambazoglu and Dr. Fabrizio Silvestri for our collaboration and for showing me how to stay positive, focused and motivated independent of circumstances, time and place. This experience helped me to evolve, not only as a researcher, but also as a person.

Moreover, I want to thank the evaluation committee, Prof. Alistair Moffat, Dr. Christina Lioma and Prof. Kjell Bratsbergsengen, reviewers and chairs of the conferences, and everyone who helped to improve the introduction part of this thesis (Svein Erik, Øystein, Magnus, Robert, Naimdjon and Barla).

Last but not least, I thank my family and friends for their support, encouragement, patience and jokes, which always helped me to see the light. Without your trust, love and friendship I would never achieve this much.

# Contents


**1 Introduction** **1**
- 1.1 Motivation . . . 1
- 1.2 Query Processing in Distributed Search Engines . . . 2
- 1.3 Research Questions . . . 4

**2 Background** **5**
- 2.1 Query Processing Basics . . . 5
  - 2.1.1 Relevance Model . . . 6
  - 2.1.2 Index Representation . . . 6
  - 2.1.3 Posting List Compression . . . 8
  - 2.1.4 Query Processing . . . 10
  - 2.1.5 Post-processing . . . 12
  - 2.1.6 Online and Batch Query Processing . . . 13
- 2.2 Query Processing Optimizations . . . 13
  - 2.2.1 Caching . . . 14
  - 2.2.2 Partial Query Processing and Skipping . . . 14
  - 2.2.3 Parallelization . . . 16
- 2.3 Distributed Query Processing . . . 17
  - 2.3.1 Index Partitioning, Replication and Tiering . . . 17
  - 2.3.2 Partitioned Query Processing . . . 18
    - Term-wise vs Document-wise Partitioning . . . 18
    - Hybrid Partitioning . . . 20
    - Pipelined Query Processing . . . 20
    - Load Balancing . . . 21
- 2.4 Performance Efficiency and Evaluation . . . 22

**3 Research Summary** **23**
- 3.1 Formalities . . . 23
- 3.2 Publications and Research Process . . . 24
- 3.3 Included Papers . . . 26
  - 3.3.1 Paper A.II . . . 26
  - 3.3.2 Paper A.III . . . 27
  - 3.3.3 Paper A.IV . . . 28
  - 3.3.4 Paper B.I . . . 29

3.3.5 Paper B.III . . . 30
3.3.6 Paper C.I . . . 31
3.3.7 Paper C.II . . . 32
3.4 Other Papers . . . 33
3.4.1 Paper A.I . . . 33
3.4.2 Paper A.V . . . 34
3.4.3 Paper A.VI . . . 35
3.4.4 Paper B.II . . . 36
3.5 Frameworks and Data . . . 37
3.5.1 Frameworks . . . 37
SolBrille . . . 37
Dogsled (Terrier) . . . 38
Laika . . . 39
Simulators . . . 40
3.5.2 Data . . . 41
Public data . . . 41
Commercial data . . . 41
Other . . . 42
3.6 Evaluation of Contributions . . . 42
3.6.1 Partitioned Query Processing . . . 42
3.6.2 Efficient Query Processing and Pruning . . . 43
3.6.3 Caching . . . 44
3.6.4 Connecting the Dots: Towards an Efficient Distributed Search Engine . . . 44
3.7 Conclusions and Further Work . . . 44
3.8 Outlook . . . 45

**Bibliography** **47**

**Included Papers** **61**

**A Paper A.II: A Combined Semi-Pipelined Query Processing Architecture for Distributed Full-Text Retrieval** **63**
A.1 Introduction . . . 65
A.2 Related Work . . . 66
A.3 Preliminaries . . . 66
Space-Limited Adaptive Pruning . . . 67
Non-Pipelined Approach . . . 67
Pipelined Approach . . . 67
A.4 A Combined Semi-Pipelined Approach . . . 68
A.4.1 Semi-Pipelined Query Processing . . . 68
A.4.2 Combination Heuristic . . . 72
A.4.3 Alternative Routing Strategy . . . 73
A.5 Evaluation . . . 74
Semi-Pipelined Execution . . . 76
Decision Heuristic . . . 76

Alternative Routing Strategy . . . 76
Combination of the Methods . . . 77
A.6 Conclusions and Further Work . . . 78
A.7 References . . . 78

**B Paper B.I: Efficient Compressed Inverted Index Skipping for Disjunctive Text-Queries 81**
B.1 Introduction . . . 83
B.2 Related Work . . . 84
B.3 Index Organization and Skipping . . . 85
B.3.1 Basic Inverted Index . . . 85
B.3.2 Self-skipping Index . . . 86
B.4 Query Processing Methods . . . 87
B.4.1 Term-At-A-Time Processing . . . 88
B.4.2 Document-At-A-Time Processing . . . 89
B.5 Evaluation . . . 90
B.5.1 Term-at-a-Time Processing . . . 90
B.5.2 Document-at-a-Time Processing . . . 92
B.5.3 Physical Block Size . . . 92
B.6 Conclusions and Further Work . . . 95
B.7 References . . . 95

**C Paper A.III: Improving the Performance of Pipelined Query Processing with Skipping 97**
C.1 Introduction . . . 99
C.2 Related work . . . 99
C.3 Preliminaries . . . 101
C.4 Improving pipelined query processing with skipping . . . 102
C.5 Max-Score optimization of pipelined query processing . . . 104
C.6 Max-Score optimization of term assignment . . . 106
C.7 Experimental results . . . 106
C.8 Conclusions and further work . . . 111
C.9 References . . . 113

**D Paper A.IV: Intra-Query Concurrent Pipelined Processing For Distributed Full-Text Retrieval 115**
D.1 Introduction . . . 117
D.2 Related work . . . 117
D.3 Preliminaries . . . 119
D.4 Intra-query parallel processing . . . 121
D.4.1 Query parallelism on different nodes . . . 121
D.4.2 Sub-query parallelism on a single node . . . 123
D.5 Experiments . . . 124
D.6 Conclusion and further work . . . 129
D.7 References . . . 130

**E Paper B.III: Improving Dynamic Index Pruning via Linear Programming 133**
E.1 Introduction . . . 135
E.2 The Max-Score Technique . . . 136
E.3 Linear Programming Solution . . . 136
E.4 Experimental Results . . . 136
E.5 Conclusions . . . 137
E.6 References . . . 137

**F Paper C.I: Modeling Static Caching in Web Search Engines 141**
F.1 Introduction . . . 143
F.2 Related Work . . . 144
F.3 Data Characteristics . . . 145
F.4 Modeling the Cache . . . 146
F.5 Experimental Validation . . . 148
F.6 Conclusions . . . 150
F.7 References . . . 152

**G Paper C.II: Prefetching Query Results and its Impact on Search Engines 155**
G.1 Introduction . . . 157
G.2 Preliminaries . . . 158
G.3 Offline Strategies . . . 162
G.3.1 Offline Query Selection . . . 162
G.3.2 Offline Query Ordering . . . 162
G.4 Online strategies . . . 163
G.4.1 Online Selection . . . 164
G.4.2 Online Ordering . . . 166
G.5 Data and Setup . . . 166
G.6 Experiments . . . 170
G.7 Previous Work . . . 175
G.8 Conclusions and Further Work . . . 178
G.9 Acknowledgments . . . 178
G.10 References . . . 179

**Additional Material 181**

**H A Retrospective Look at Partitioned Query Processing 183**

# Chapter 1

# Introduction

*"Computers are useless.*
*They can only give you answers."*

– Pablo Picasso

This thesis is a collection of papers bound together by a general introduction. This chapter presents the motivation and the context of our work and overviews the research questions. Chapter 2 summarizes the background knowledge and related work. The research summary and an evaluation of the contributions are given in Chapter 3, which also reviews the papers that were written as a part of this research but not included in the final thesis, outlines several directions for future work, and concludes the thesis. The included papers can be found in the Appendix.

## 1.1 Motivation

Since 1993, the World Wide Web has experienced a tremendous growth. In May 2012, the indexed part of the Web was estimated to contain at least 8.21 billion pages.[1] In June 2011, the sites of Facebook, Microsoft and Yahoo! alone were estimated to have more than half a billion unique visitors each, and the sites of Google had already passed the 1 billion mark.[2] With Web 2.0, the interaction between the users and the content became even more critical. Today, it is expected that information becomes immediately available to millions of users and that each user can instantly reach the most recent and related information.

Traditionally, keyword-based search has been the main tool for finding and accessing information on the Web. It is also one of the essential components in most of the online applications. For example, Flickr (Yahoo!) uses keyword search for finding pictures in

[1] `http://www.worldwidewebsize.com`, The size of the World Wide Web, visited May 3, 2012.

[2] `http://www.comscoredatamine.com`, Google Reaches 1 Billion Global Visitors, June 2011, visited on May 3, 2012.

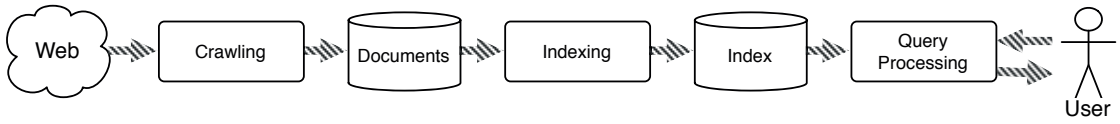



Figure 1.1: Simplified information flow in a Web search engine.

an enormous archive of publicly available photos, and YouTube (Google) does the same for a gigantic archive of videos. Google, Microsoft and Yahoo! use search to filter and organize private emails, select specific content, such as news and media, and target online advertisements. Amazon and eBay use search to explore and target products in their online mega-stores. Google, Microsoft, and now Amazon, offer companies enterprise search for their public and internal use. Social search, search in government documents, patent, legal and medical search are only a few of the many other applications.

The growth in the amount of information available for search, rapidly increasing usage and tight performance constraints have driven companies to build large, geographically distributed data centers housing thousands of computers, consuming enormous amounts of electricity and requiring a huge infrastructure around. Companies like Google annually spend billions of U.S. dollars on building and running their data centers.[3] At this scale, even minor efficiency improvements may result in large financial savings.

In this thesis, we elaborate on efficient query processing and address several problems within partitioned query processing, pruning and caching. We propose several novel techniques and evaluate them either using a real implementation or by simulation with help of real-world data. In general, our results indicate significant improvements over the state-of-the-art baselines and thus can contribute to reducing infrastructure costs and to improving user satisfaction.

## 1.2 Query Processing in Distributed Search Engines

In this section, we describe the context of our work. A search engine consists of three main components: a crawler, an indexer and a query processor. Additional components can be used for specific tasks, such as spam detection, page ranking and snippet generation. The crawler follows links between pages and downloads the content for further processing. The indexer processes the downloaded content in a specific way and stores it in a certain fashion optimized for further processing. The query processor receives queries from the user, pre-processes them, matches them against the index, ranks the results and presents them to the user. Figure 1.1 illustrates the information flow.

Cambazoglu and Baeza-Yates [43] have presented a comprehensive survey of scalability challenges in modern Web search engines. In this survey, the three tasks described above (crawling, indexing and query processing) are combined with four types of granularity:

[3] `http://www.datacenterknowledge.com`, Google Spent $951 Million on Data Centers in 4Q, January 2012, visited on May 3, 2012.

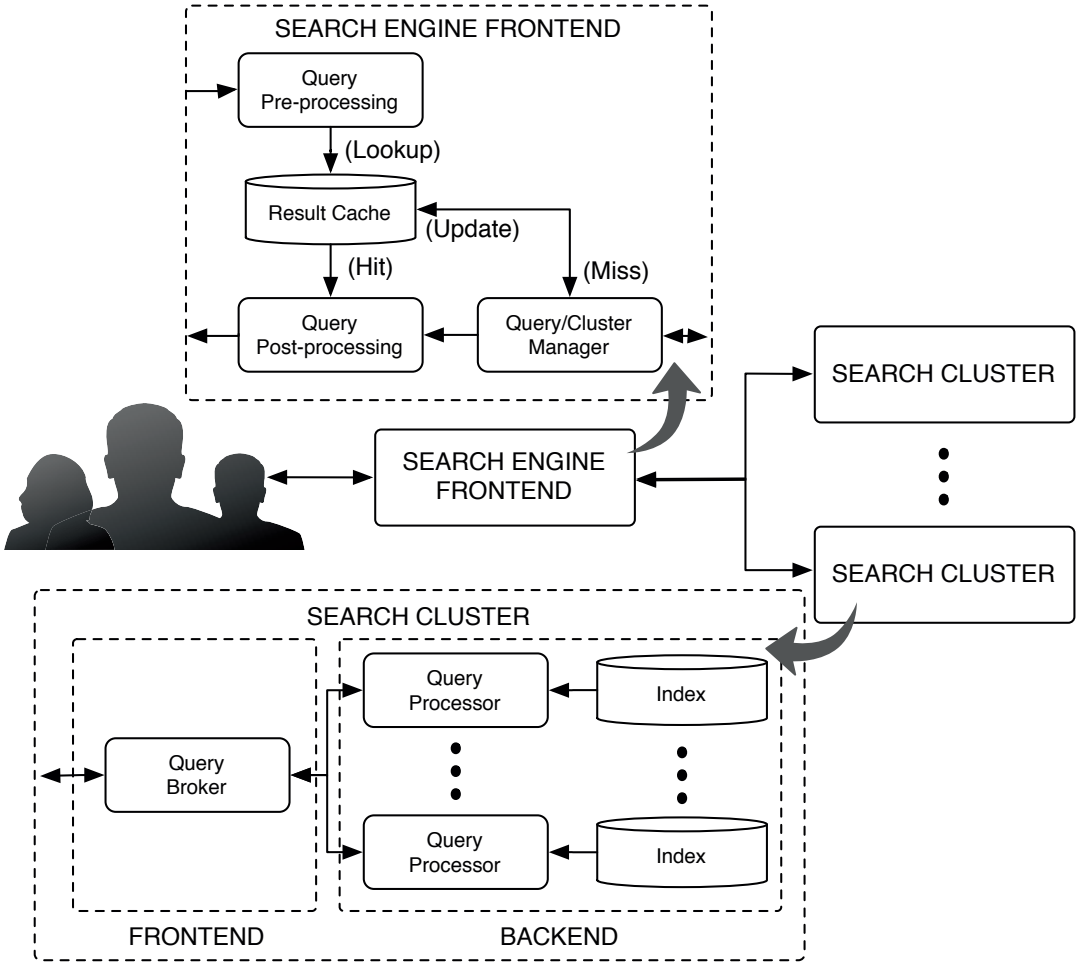



Figure 1.2: Query processing architecture addressed in this thesis.

single node, multi-node cluster, multi-cluster site and multi-site engine. According to this classification, the challenges addressed in this thesis are associated mainly with query processing with single and multi-node granularity. As being related to query processing, we also address index organization and representation. Among the challenges listed by Cambazoglu and Baeza-Yates, we directly address caching, index compression, early termination, index partitioning, load balancing and query degradation. We explain these challenges and some of the existing solutions in the next chapter. A detailed description of our papers and contributions follows in Chapter 3.

In Figure 1.2, we illustrate a generalized architecture addressed in our work. The search engine consists of a frontend and one or several clusters. The frontend and the clusters can be either on the same site or on several geographically distributed sites. The frontend is responsible for the interaction with the user and tasks such as query pre- and post-processing, deciding which cluster to use and result caching, while the processing itself is done by the clusters. For simplicity, we look at a system with only one cluster. The cluster itself consists of a frontend and a backend. The query broker is responsible for receiving queries from the search engine frontend and scheduling these to the cluster backend, which

is a set of independent nodes doing the actual processing. The broker may further be responsible for tasks such as load balancing and result aggregation within the cluster.

## 1.3 Research Questions

The main question addressed in this thesis is:

**RQ** *What are the main challenges of existing methods for efficient query processing in distributed search engines and how can these be resolved?*

Seeking to answer this question, the research is organized as an exploratory, iterative process consisting of observation or literature study in order to define a problem or to state a hypothesis, proposal of a solution, quantitative evaluation against the baseline and publication of the results. The research and the contributions within this thesis are divided in three main directions:

**RQ-A** *How to improve the efficiency of partitioned query processing?*

**RQ-B** *How to improve the efficiency of query processing and pruning?*

**RQ-C** *How to improve the efficiency of caching?*

Consequently, the logical naming of the papers written in connection with this thesis represents the research direction and the chronological order. For example, Paper C.II is our second paper on caching. An overview of the papers and the research process is given in Sections 3.2 and 3.3 and the total contribution of the thesis is evaluated in Section 3.6. Finally, we remark that Papers A.I, A.V, A.VI and B.II are not included in this thesis. However, because they were written in connection with this thesis, we describe the methodology and the research process behind them. This description can be found in Section 3.4.

# Chapter 2

# Background

*"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."*

– Stan Kelly-Bootle

In order to ease the understanding of our work and contributions, this chapter gives an overview of the technical background and related work. In Section 2.1, we explain the query processing basics including similarity models, inverted indexes and index compression. Section 2.2 covers the query processing optimizations addressed in our work. These include caching, partial processing, skipping and parallelization. Distributed query processing is discussed in Section 2.3, which covers index partitioning and other techniques, such as replication and tiering. Finally, in Section 2.4, we briefly discuss efficiency indicators and evaluation methodology. Further and more comprehensive knowledge can be obtained from several recent books [24, 40, 92, 102], surveys [43, 166] and other cited literature.

## 2.1 Query Processing Basics

We look at the task of finding the $k$ most relevant documents for a given textual query $q$. As specified in the previous chapter, this task involves an indexing phase and a query processing phase. Additionally, in order to support document updates, indexing can be repeated at certain intervals. In our work, we do not address index creation and updates, and focus mainly on the query processing phase. Furthermore, both queries and documents are preprocessed in a similar fashion, which involves several subtasks such as term extraction and normalization (e.g., character normalization and removal, stop-word removal and stemming). In this case, a query processor's task is to match the normalized query terms against a previously constructed index over the terms in the documents.

### 2.1.1 Relevance Model

In our work, both queries and documents correspond to bags of keywords and the result set of a query represents those $k$ documents that maximize the degree of estimated relevance. In this case, query processing involves matching query terms against the indexed documents and estimation of their relevance scores. There exist a large number of relevance models and ranking functions, which can be found in books [24, 40, 102, 166] and related literature. In our work, we consider a variant [40] of the Okapi BM25 model [131], where the relevance of a document $d$ to a query $q$ is measured as follows:

$$\text{Score}(d, q) = \sum_{t \in q} w_{t,q} \cdot w_{t,d} \tag{2.1}$$

$$w_{t,q} = \frac{(k_3 + 1) \cdot f_{t,q} / \max(f_{t,q})}{k_3 + f_{t,q} / \max(f_{t,q})} \tag{2.2}$$

$$w_{t,d} = \text{TF}(t, d) \cdot \text{IDF}(t) \tag{2.3}$$

$$\text{TF}(t, d) = \frac{f_{t,d} \cdot (k_1 + 1)}{f_{t,d} + k_1 \cdot ((1 - b) + b \cdot l_d / l_{avg})} \tag{2.4}$$

$$\text{IDF}(t) = \log(\frac{N - f_t + 0.5}{f_t + 0.5}) \tag{2.5}$$

Here, $w_{t,q}$ and $w_{t,d}$ measure the importance of a query term $t$ to $q$ and $d$ respectively. Further, $w_{t,q}$ is computed from the number of occurrences of $t$ within $q$, $f_{t,q}$ (*query frequency*), while $w_{t,d}$ is computed as a product of the term frequency (TF) and inverse document frequency (IDF) functions. Subsequently, TF uses the number of occurrences of $t$ in $d$, $f_{t,d}$ (*term frequency*), the number of tokens contained in $d$, $l_d$ (*document length*), and the average document length within the indexed collection, $l_{avg}$. IDF uses the number of documents in the indexed collection, $N$, and the number of indexed documents containing $t$, $f_t$ (*document frequency*). Additionally, the model uses three system-dependent constants with default values $k_1 = 1.2$, $k_3 = 8$ and $b = 0.75$.

**Other alternatives.** Alternative ranking functions may consider use of link-analysis methods such as SALSA, HITS and PageRank [92, 96, 102] or other forms of global page scores [97, 134, 161], term positions and proximity [40, 71, 156, 164], geographical proximity and coordinates [50, 53, 54, 57, 159, 160], or combinations thereof.

### 2.1.2 Index Representation

For efficient query processing, search engines traditionally deploy *inverted indexes* [114, 154, 166]. For each indexed term $t$, the inverted index stores a sequence of IDs of the documents in which $t$ appears and the corresponding number of occurrences. Additionally, the inverted index may store positions at which a term appears within a given document or other contextual information.

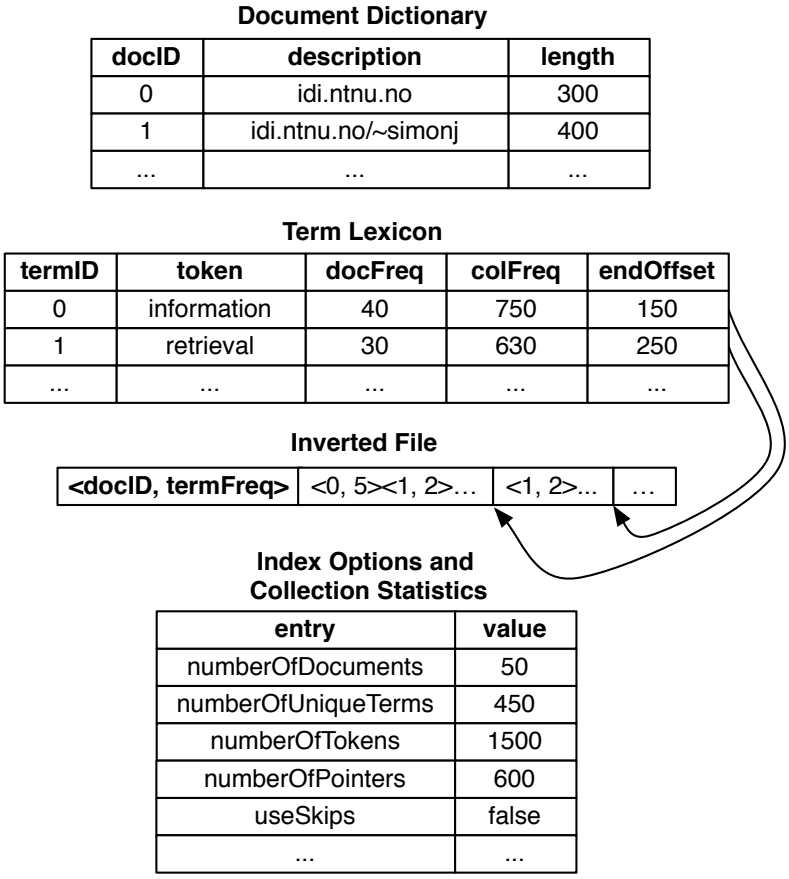



Figure 2.1: Basic index organization with example data.

Figure 2.1 illustrates the basic index organization used in our work. The index consists of four data structures: *document dictionary*, *term lexicon*, *index options and collection statistics*, and finally the *inverted file* itself. The document dictionary maps document IDs to document descriptions and additional information, such as document length in number of tokens. The term lexicon maps term IDs to their token representation, document and collection frequencies, and the end-offset within the inverted file. The options and statistics file contains collection statistics such as the number of indexed documents, unique terms, tokens and pointers in the index. For the inverted file itself, we consider a document-ordered posting list representation, where each term is represented by a sequence of postings ordered by increasing document ID. Furthermore, we consider both lexicon and document dictionaries to be represented as constant length, ID-sorted arrays and accessed using binary search.

**Other alternatives.** A number of publications look at frequency or impact-ordered inverted lists [4, 6, 8, 9, 123, 124, 142], bitmap indexes [71, 154], wavelet trees [117], various combinations of trees and inverted indexes [54, 57, 159, 160], column stores and score materialization [77]. Moreover, the lexicon can be efficiently represented using a suffix array, and the document dictionary can be implemented as a $B^+$ tree or similar.

Table 2.1: Compression example for Unary and Elias-$\gamma$ codes [154].

| Value | Unary | Elias-$\gamma$ |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 10 | 10 0 |
| 3 | 110 | 10 1 |
| 4 | 1110 | 110 00 |
| 5 | 11110 | 110 01 |
| 6 | 111110 | 110 10 |
| 7 | 1111110 | 110 11 |
| 8 | 11111110 | 1110 000 |
| 9 | 111111110 | 1110 001 |
| 10 | 1111111110 | 1110 010 |

### 2.1.3 Posting List Compression

Posting list compression has been discussed in many books and papers. According to Shannon, the length of an optimal code representation of an entry with probability $P(x)$ is $-\lceil \log_2 P(x) \rceil$ bits [154]. Further, it is known that term frequencies follow a power law probability distribution [165]. These two factors have motivated variable-length compression of posting frequencies. Additionally, a sequence of increasing document IDs can be represented using differences between the consecutive entries, called *deltas* or *gaps*. In other words, $i$-th document ID, $d_i$, can be represented as $d_i - d_{i-1}$. As deltas are naturally smaller than the original values and smaller deltas are more probable than larger ones, this leads to an efficient representation.

In the following, we briefly describe the compression methods used in our work. For other methods and recent trends, we refer to the related work.

**Unary and Elias-$\gamma$.** Traditionally, search engines have used bit-aligned variable-length methods. The simplest of these is Unary [154], which compresses an integer $x$ as $x-1$ one-bits followed by a single zero-bit. This method is perfect when the probability distribution of $x$ is $P(x) = 2^{-x}$ (for $x > 0$). With a less skewed distribution, this method is highly expensive for representing large values. Elias-$\gamma$ [154] improves compression by encoding $1 + \lfloor \log_2 x \rfloor$ in Unary followed by $x - 2^{\lfloor \log_2 x \rfloor}$ encoded in binary. In this case, the ideal probability distribution is $P(x) = 2^{-(1+2\log_2 x)}$. We illustrate both methods in Table 2.1.

**VByte.** Compression with bit-aligned codes requires many arithmetical and bitwise operations, and therefore is very time-consuming. For performance reasons, byte-aligned codes are a more recent trend. The simplest of these methods is VByte [153], which splits the binary representation of an integer $x$ into several codewords of seven bits. The most significant bit of each codeword is then set to one if it is followed by more codewords, or zero otherwise. Both compression and decompression with VByte require one branch condition and only a few byte operations per codeword. However, the method wastes unused bits (e.g., $x = 1$ requires one byte).

**Simple-9.** More recent publications look at word-aligned methods, which combine the decompression speed of VByte with an improved compression ratio. One such method is

Table 2.2: Simple-9 coding schemes [5].

| Selector | Number of codes | Code length (bits) |
|---|---|---|
| a | 28 | 1 |
| b | 14 | 2 |
| c | 9 | 3 |
| d | 7 | 4 |
| e | 5 | 5 |
| f | 4 | 7 |
| g | 3 | 9 |
| h | 2 | 14 |
| i | 1 | 28 |

Simple-9 [5], which represents a sequence of up to 28 numbers with a binary codeword of 32 bits. A codeword consists of four selector bits and 28 payload bits. As we illustrate in Table 2.2, the selector bits allow nine different combinations. The payload may contain 28 one-bit values, 14 two-bit values or etc., where the number of bits per value is determined by the largest value in the sequence.

**NewPFoR.** The most recent methods try to optimize compression and especially decompression for modern CPU architectures. One of the methods, PFoR [167], has received a lot of attention. PFoR is an extension of the Frame of Reference (FoR) method, and means Patched FoR.[1] With FoR, each integer $x$ in a sequence (*chunk*) of values is encoded as $x - \min$, where $\min$ is the smallest value in the chunk. In this case, it requires $\lceil \log_2(\max - \min - 1) \rceil$ bits to encode each value and an additional codeword to represent the header of the chunk. In order to further optimize the compression ratio, PFoR encodes each value in the chunk using only $b$ bits. Values that require more than $b$ bits are called *exceptions*. In order to represent these, the header stores an offset to the first exception and each consecutive exception represents the offset to the next exception. The exception values themselves are stored at the end of the block. Furthermore, the original work [167] suggests encoding in chunks of 128 entries and choosing $b$ to satisfy 90% of them.

A major drawback of PFoR is that it has to force one false exception after each $2^b$ encoded values. In order to overcome this problem, Yan et al. [157] have introduced a method called NewPFoR. With NewPFoR, the $b$ least significant bits of each entry are stored in the main block and the overflow bits of exceptions are stored as two Simple-9 encoded arrays (one for the offsets and one for the overflow bits). In Figure 2.2, we illustrate the layout of a block encoding 128 entries.

These methods relate to our work as follows. In our early work (Paper A.II), we use Unary codes for frequencies and Elias-$\gamma$ for deltas. Our more recent work (Papers A.III, A.IV, A.VI, B.I and B.III) applies NewPFoR compression to frequencies, document ID deltas and skip offsets. In our case, we use chunks of 128 entries and chunks with less than 100 entries are compressed with VByte. Further details can be found in our papers.

[1] PFoR is often abbreviated as PForDelta, PFD or P4D. For the sake of generality, we assume delta-extraction to be an additional step and not a part of the method itself.

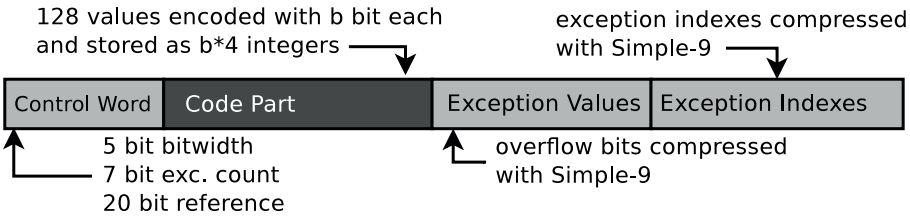



Figure 2.2: Data layout of a chunk compressed with NewPFoR.

**Other alternatives.** For a more complete description of the Unary and Elias methods, we refer the reader to the classic book by Witten et al. [154], which explains these and more advanced bit-aligned compression methods (including Elias-$\delta$, Golomb, Rice and Interpolative codes), signature files and bitmaps. Further, as it can be seen from Figure 2.2, Simple-9 leaves some of the payload bits unused and uses four bits to represent nine selector values. For this reason, several alternative schemes (such as Simple-16 [162] and Relative and Carryover families [5]) have been proposed. For more information on these methods, we refer to the work by Anh and Moffat [5, 7] and subsequent publications, also including methods for 64-bit words [10]. In the same paper with NewPFoR, Yan et al. [157] have proposed a method that trades-off compression ratio and decompression speed, OptPFoR, and several techniques that optimize compression of frequencies. Compression of term positions has been addressed in another paper by Yan et al. [156] and related publications. Moreover, several publications have studied document ID reordering [135, 157] and posting list pairing [91] in order to improve compression even further. Finally, several recent publications have considered adaptive frame-of-reference methods [59, 137].

### 2.1.4 Query Processing

With the constraints stated in Section 2.1.1, the query processor's task is to iterate through the postings lists corresponding to the query terms, score their postings, accumulate scores of candidate documents, find the IDs of the $k$ highest scored documents and finally sort them according to their score and return them to the user. Query processing can be *term-at-a-time* or *document-at-a-time*. With the term-at-a-time (TAAT) approach, the query processor has to evaluate the posting list of a term completely before any other term is considered. For this reason, the query processor has to keep scores of partially evaluated documents, called *accumulators*. After processing the last term, the query processor has to extract the $k$ best candidates from the scored accumulators, which is usually done with the help of a $k$-entry minimum heap. We illustrate the process in Algorithm 2.1.

With the document-at-a-time (DAAT) approach, the posting lists of different terms are evaluated in parallel. Each iteration of the query processor picks a candidate document ID (the least of the current document IDs referred by the iterators), accumulates its score from the corresponding iterators and advances these iterators to the next posting. Finally, a fully scored candidate is inserted into the heap when its score is larger than the score of the current $k$-th best candidate. We illustrate the process in Algorithm 2.2.

**Algorithm 2.1**: Term-at-a-time query processing (Full TAAT)

**Data**: posting list iterators $\{i_1, \ldots, i_l\}$ sorted by ascending collection frequency $F_t$
**Result**: $k$ best query results sorted by descending score

```
1 A ← ∅;
2 foreach iterator i_t do
3     while i_t has more postings do
4         d ← i_t.d();
5         if ⟨d, s⟩ ∈ A then s ← s + i_t.s();
6         else A ← A ⋃ ⟨d, i_t.s()⟩;
7         i_t.next();
8 return resHeap.decrSortResults(A);
```

**Algorithm 2.2**: Document-at-a-time query processing (Full DAAT)

**Data**: posting list iterators $\{i_1, \ldots, i_l\}$
**Result**: $k$ best query results sorted by descending score

```
1 resHeap ← ∅;
2 while {i_1, ..., i_l} ≠ ∅ do
3     d ← min(i_{t∈{1,...,l}}.d()); s ← 0;
4     foreach iterator i_t s.t. i_t.d() = d do
5         s ← s + i_t.s();
6         if i_t.next() = false then close and remove i_t;
7     if s > resHeap.minScore then resHeap.insert(d, s);
8 return resHeap.decrSortResults();
```

TAAT can be considered as more efficient with respect to index access (especially for disk-based indexes), buffering, CPU cache and compiler optimizations. However, it has to maintain a complete accumulator set, which at the end is equivalent to a union of the posting lists. On the other hand, DAAT requires parallel access to posting lists, which affects the performance of the traditional hard-disks and internal CPU cache, but it has to keep only the $k$ best candidates seen so far.

Furthermore, the methods described so far assume that documents matched by *any* of the query terms may be reported as a result. Queries with such semantics are called *disjunctive* or OR queries. A common technique to reduce the query processing cost is to assume that documents matched only by *all* of the query terms can be returned as a result. Such queries are called *conjunctive* or AND queries. The modified versions of Algorithms 2.1 and 2.2 are provided in Algorithms 2.3 and 2.4. In these algorithms, $i_t.skipTo(d)$ advances the iterator to the first posting with document ID equal to or larger than $d$.

As the algorithms show, AND TAAT benefits from processing the shortest posting list first and then matching the accumulator set against the postings of other terms. In this case, the accumulator set is never larger than the size of the shortest posting list. AND DAAT benefits from using the maximum document ID among those currently being referred by the iterators. In this case, most of the postings in all lists can be skipped. For performance

**Algorithm 2.3**: AND TAAT

**Data**: posting list iterators $\{i_1, \ldots, i_l\}$ sorted by ascending collection frequency $F_t$
**Result**: $k$ best query results sorted by descending score

```
1 A ← ∅;
2 foreach posting in i_1 do A ← A ⋃ ⟨i_1.d(), i_1.s()⟩;
3 foreach iterator i_t ∈ {i_2, ..., i_l} do
4     foreach accumulator ⟨d, s⟩ ∈ A do
5         if i_t.d() < d then
6             if i_t.skipTo(d) = false then A ← A − {⟨d', s'⟩ s.t. d' ≥ d} and proceed to the
              next iterator;
7         if i_t.d() = d then s ← s + i_t.s();
8         else A ← A − ⟨d, s⟩;
9 return resHeap.decrSortResults(A);
```

reasons, conjunctive queries are often used in practical search engines. However, the result sets returned by AND queries are potentially smaller than those returned by OR queries. This might be a problem, especially when one of the terms is missing or mistyped. For this reason, a naive solution is to evaluate a query in the AND mode, and if the results are not sufficient, evaluate it in the OR mode. However, better performance can be achieved by partial query processing, which we explain in Section 2.2.2.

**Other alternatives.** There exist a large number of alternative query processing methods, depending on the query processing task and posting list ordering. For example, matching of query terms can be extended to Boolean [9] and Complex Boolean [143] queries, wildcard or phrase queries. Furthermore, a number of publications [15, 25, 26, 60, 144] consider performing set intersection on postings lists. Some of these publications [144] consider performing an intersection of posting lists first, and then score and rank only the documents that are contained in the intersection result set. In a recent work, Culpepper and Moffat [56] have presented an efficient approach to posting list intersection that applies a hybrid combination of posting lists and bit-vectors. Finally, frequency and impact-ordered lists lead to a slightly different query processing approach and further optimizations.

### 2.1.5 Post-processing

Once a query has been processed, the search engine usually performs a post-processing step where it converts document IDs into document descriptions/previews called *snippets*. A snippet normally represents a fragment of the text that has the most accurate match with the query. Additionally, the system may track the actions performed by the user on the result pages in order to re-run a modified query or to adjust the global page scores.

**Algorithm 2.4**: AND DAAT

```
Data: posting list iterators {i_1, ..., i_l}
Result: k best query results sorted by descending score
resHeap ← ∅;
while true do
    d ← max(i_{t∈{1,...,l}}.d());
    foreach iterator i_t s.t. i_t.d() < d do
        if i_t.skipTo(d) = false then break the main loop;
    if ∀i_t(i_t.d() = d) then
        s ← 0;
        foreach iterator i_t do
            s ← s + i_t.s();
        if s > resHeap.minScore then resHeap.insert(d, s);
        foreach iterator i_t do
            if i_t.next() = false then break the main loop;
return resHeap.decrSortResults();
```

### 2.1.6 Online and Batch Query Processing

In general, it is possible to separate between *online* and *batch* (or *offline*) query processing. With the former approach, queries are submitted one at a time, with certain regularity and the results have to be returned as soon as possible. Such queries have many interesting properties [22, 122, 141, 146, 158] regarding the number of queries per user, repetition and modification of queries, number of pages viewed and actual clicks, geography, locality, etc. Additionally, the online query traffic to a local search engine varies during a day – it is high at the afternoon and low at the night/morning [28]. Each search system (cluster, site or the engine) has a peak sustainable throughput (PST) that it can support. If the query arrival rate goes above PST, the waiting queue starts to grow indefinitely. In order to avoid starvation, the search engine may *degrade* some of the queries.

Differently from online queries, batch queries [62] do not have such strong processing constraints. These can be easily delayed or reordered in order to optimize processing and executed at the rate of PST.

Moreover, it is possible to have different combinations of these. For example, online queries can be grouped in tiny (millisecond-level) batches. Alternatively, online queries can be augmented with batch queries. In Paper C.II, we show that the latter scenario can improve the overall performance of a Web search engine.

## 2.2 Query Processing Optimizations

This section overviews three types of query processing optimizations addressed in our work. Section 2.2.1 overviews caching, which is addressed in Papers C.I and C.II. Sec-

tion 2.2.2 overviews skipping and pruning, which are addressed in Papers A.III, B.I and B.III. Section 2.2.3 overviews query processing parallelization on modern multi-core processors; intra-query concurrency is addressed in Paper A.IV.

### 2.2.1 Caching

*Caching* refers to reuse of previously computed data or storage of frequently accessed data higher in the memory hierarchy, and it can be applied at several levels of query processing. Because queries follow a power-law distribution (frequencies) and have temporal (submission time) and spatial (requested pages) locality, early work in this direction has been concentrated on result caching [93, 109, 158]. Later work led to two, three and, most recently, five-level caches. A two-level cache [133] consists of a result and a posting list cache, while a three-level cache [98] has an additional intersection cache for the most frequent term pairs and a five-level cache [119] additionally has a document cache and a score cache. Separate document ID and HTML caches for results have been proposed [3]. Additionally, distributed search engines may have use of location caches [103] and more advanced caching mechanisms [106].

In parallel with the evolution of the cache hierarchy, a lot of discussion has emerged around static and dynamic caches [17, 18, 67, 68, 120, 136], caching and index pruning [138, 148], and replacement [67, 68, 93, 125] and admission [21] policies. Some of these publications [68, 93, 95, 136] consider prefetching of consecutive pages and documents that are likely to be requested. More recent publications optimize the actual processing cost rather than the hit ratio [75, 121], look at a combination of posting list compression and caching [162], and present several cache optimizations for batch query processing [62].

Finally, several of the most recent publications assume an infinite result cache [45] and focus on the result freshness [1, 2, 29, 35, 45]. In this case, cached results can be associated with a time-to-live (TTL) value, which can be either fixed for all queries or chosen for each query result independently, and several mechanisms can be used to deal with invalidation and refreshment of old or stale entries.

### 2.2.2 Partial Query Processing and Skipping

*Partial query processing* is also known as *dynamic pruning* and *early exit optimizations*. It contains a great variety of query processing optimizations that try to minimize the processing cost by reducing the number of postings being scored. Some of the early optimizations can be found in the work by Buckley and Lewit [38]. Ten years later, Turtle and Flood [150] discussed these and several other early ideas, described the Max-Score heuristic and evaluated it for both TAAT and DAAT query processing. The idea behind Max-Score is to eliminate the candidates that cannot enter the result set, i.e., if the current score of a candidate (accumulator) plus the maximum impact that it can achieve from the not yet scored postings is less than the score of the current $k$-th best candidate.

Nearly at the same time, Moffat and Zobel [113, 114] presented the *Quit* and *Continue* heuristics and explained the use of skips. The Quit approach suggests to stop query pro-

cessing once the target number of accumulators is reached, while the Continue approach proceeds increasing scores of the existing accumulators.

A performance comparison between Max-Score and Quit/Continue methods has been presented by Lacour et al. [90]. The work on Max-Score has been advanced by Strohman et al. [143], who further optimized pruning efficiency for complex queries by placing a segment containing the top ranked documents in the beginning of each posting lists. Lester et al. [94] have presented an efficient space-limited pruning method, which is based on dynamic frequency and score thresholds, and according to the presented results outperforms the Quit/Continue methods. Broder et al. [37] have presented an interesting and efficient approach, widely known as WAND, which maintains posting list iterators ordered by current document ID and then uses the minimum required score to choose a *pivot*. Then, if the first and the pivot iterators correspond to the same document ID, it performs an evaluation step. Otherwise, it advances one of the iterators and repeats pivot selection.

WAND and Max-Score are indeed two very similar but different methods. Most recently (at the time of our own work), Ding et al. [65] have presented an efficient extension of WAND called Block-Max, while Fontoura et al. [72] have presented an interesting study of Max-Score and WAND used in a large-scale commercial advertising framework. Finally, a follow-up publication by Shan et al. [134] presented extensions of both Max-Score and WAND, this time optimized for a combination of Block-Max indexes and global page scores (e.g., PageRank). We finally remark that WAND and Max-Score are *safe* optimizations, because they guarantee the same results as a full evaluation, whereas Quit/Continue and the method by Lester et al. are *unsafe*.[2]

**Skipping.** According to Moffat and Zobel [114], *skips* are synchronization points in compressed posting lists and these can be represented as a full document ID and the offset to the next skip entry:

**Original postings:**

$$\langle 5,1\rangle\langle 8,1\rangle\langle 12,2\rangle\langle 13,3\rangle\langle 15,1\rangle\langle 18,1\rangle\langle 23,2\rangle\langle 28,1\rangle\ldots$$

**Encoded postings:**

$$\langle\langle 5,o_2\rangle\rangle\langle 5,1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle\langle 13,o_3\rangle\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,1\rangle\langle\langle 23,o_4\rangle\rangle\langle 5,2\rangle\langle 5,1\rangle\ldots$$

Figure 2.3: Illustration of simple embedded skip pointers [114].

Moffat and Zobel have shown how to calculate the optimal distance for single-level and multiple-level skips. The work on optimal skips has been followed by Boldi and Vigna [31] (skip towers) and Chierichetti et al. [51] (spaghetti skips), while Büttcher and Clarke [39] presented an efficient tree-like index organization optimized for random-access look-ups and CPU L1 and L2 caches of modern processors. However, more recent publications [65, 144] assume that posting lists can be stored in main memory, where skip pointers are stored as additional, uncompressed arrays.

[2] More precisely, an optimization can be *score-safe to* $k$, i.e., it produces the same scores as a full evaluation up to the $k$-th result, *rank-safe to* $k$, i.e., it produces the same ordering of the top $k$ results, but not necessary the same scores, or *set-safe to* $k$, i.e., it produces the same but possibly permuted top $k$ results. According to this definition Max-Score and WAND are score-safe to $k$.

**Other alternatives.** There exist a large number of alternative methods. Among these, Persin et al. [123, 124] have suggested to sort postings by frequency in order to terminate early and without using skips. Impact-ordered lists were presented in several publications by Anh and Moffat [4, 6, 8, 9] and further improved by Strohman and Croft [142]. Pruning optimizations for global page scores were considered in several publications [97, 134, 161]. Static index pruning and multi-tiering techniques were considered in the work by Carmel et al. [49] (static pruning) and Risvik et al. [129] (multi-tiering) and more recent literature. Early exit optimizations for machine learned ranking methods can be found in the work by Cambazoglu et al. [48]. Threshold Algorithm (TA), No Random Access algorithm (NRA) and similar methods, more typical for database queries, can be found in the work by Fagin et al. [66] and related literature. Some pruning and skipping optimizations can also be found in the work on set intersection methods (e.g., galloping search, Bloom-filters, hash-partitioning, etc.).

### 2.2.3 Parallelization

Several recent publications look at multi-core optimizations for IR. A mention of using commodity multi-processor computers can be found in the classic paper by Barroso et al. [27] describing the architecture of Google Web search in 2003. In a more recent publication, Tatikonda et al. [145] have presented a performance comparison between *inter-query* and *intra-query parallelism* for posting list intersection on commodity multi-core processors. Inter-query parallelism is achieved by executing different queries concurrently, while intra-query parallelism is achieved by concurrent execution of different parts of the same query. Evidently, intra-query parallelism mainly reduces the latency, while inter-query parallelism mainly improves the throughput [145]. Additionally, because of a higher parallelization overhead, intra-query concurrency is less suitable for batch query processing or processing of online queries with high arrival rates.

A recent extension of the work by Tatikonda et al. [144] shows that fine-grained intra-query parallelism for posting list intersection results in a 5.75 times speed-up on an eight-core CPU. However, the processing model in this work is limited to intersection of memory-based posting lists and skip-pointers used for task scheduling are stored as non-compressed arrays.

**Related work.** Johnson et al. [80] have presented an interesting discussion on parallelization of database queries. Tsirogiannis et al. [149] and Ding et al. [61] have looked at parallelization of set intersection queries. Frachtenberg [74] has discussed fine-grained versus coarse-grained parallelism in Web search engines. Bonacic et al. [33, 34] have looked at multi-core techniques for partitioned query processing.

Moreover, query parallelization ideas are not limited to CPUs only. Modern graphic cards provide a great amount of computation power at a relatively low price. In an early work, Ding et al. [63] have looked at using graphical processors (GPUs) for inverted index decompression and intersection of a memory-based index and reported a total speed-up of 2.37. A follow-up paper by Ding et al. [64] further discusses compression methods and parallelization techniques and shows that GPUs are quite efficient for exhaustive OR

queries, while on AND queries the difference between CPU and GPU processing is insignificant, and a combination of GPU and CPU significantly improves the performance in either case.

## 2.3 Distributed Query Processing

This section is divided into two parts. Section 2.3.1 briefly describes several methods including index partitioning, replication and tiering. Section 2.3.2 provides a more comprehensive overview of the previous work on partitioned query processing, and pipelined query processing in particular. Furthermore, our comparison between different partitioning strategies can be found in Paper A.I, while several improvements to pipelined query processing are presented in Papers A.II–A.IV.

### 2.3.1 Index Partitioning, Replication and Tiering

The work on distributed query processing started from index partitioning methods [147]. The main intention here is to distribute a large index across multiple disks or even nodes. With the introduction of Web search, it became necessary to search over a large and rapidly increasing index and to process queries from an exploded number of users. The solution proposed by Risvik et al. [130] is to combine document-wise index partitioning and replication. In this case, the query processing nodes can be viewed as a two-dimensional array, where each row corresponds to an index replica, and each column corresponds to a collection subset. Thus, by adding more columns it is possible to scale with respect to the index size, while by adding more rows it is possible to scale with respect to the query capacity. Additionally, Risvik et al. [129] have suggested to group documents into several tiers of different sizes that can be assigned to different nodes. In this case, query processing starts always at the lowest (smallest) tier, and a heuristic determines whether a higher tier has to be used or not. In combination with the architecture described in Chapter 1, partitioning, replication and tiering can be done by different clusters.

A large-scale search engine may have several geographically distributed sites. Recently, Baeza-Yates et al. [23] have presented an analysis of a two-tier system, where tiers can be geographically distributed. Next, Cambazoglu et al. [46] have shown that a multi-site architecture applying partial replication and query forwarding can lead to efficiency and effectiveness improvements, while Baeza-Yates et al. [19] have shown that a multi-site architecture improves also the operational costs when compared to a centralized one. This work is followed by the work of Cambazoglu et al. [47], who improved query forwarding between geographically distributed sites, and Brefeld et al. [36] and Blanco et al. [30], who improved document assignment. Finally, Kayaaslan et al. [89] have studied query forwarding between geographically distributed sites as a method to reduce the electricity bill. While we do not elaborate on any of these optimizations, they match the architecture described in Chapter 1 perfectly and may coexist with our own contributions.

**Other alternatives.** There exist several other directions for distributed IR. Some interesting work can be found in IR-specialized databases [55], peer-to-peer systems [139], and

federated search systems [41]. The MapReduce framework [58] is widely used for indexing [110]. Although it is less typical to be used in IR query processing, there exists work on applying it in pattern-based search, database joins, graph queries, etc.

### 2.3.2 Partitioned Query Processing

Index partitioning splits an index into several disjoint subsets and it can be done either *term-wise* or *document-wise*. Document-wise partitioning is also known as *local indexing*, because each node indexes an independent subset of the document collection, while term-wise partitioning is also known as *global indexing*. As described below, both methods have their own advantages and challenges. In order to combine the advantage of these methods, several *hybrid* partitioning schemes have been suggested. Furthermore, query processing is normally done by multi-casting a query to all of the corresponding nodes, fetching the data, potentially doing some processing, transferring their partial results to the query broker (also known as the receptionist node) and combining these in order to obtain the final results. Differently from this, *pipelined query processing* suggests to route a query through the corresponding nodes, each doing a part of processing, and finally obtain the results on the last node. Last but not least, the efficiency of both term-wise and document-wise index partition can be limited by load imbalance, which has been a topic for several publications. In the remainder of this section, we overview the ideas and the main results of several previous publications. More details can be found in our own publications and the cited literature.

#### Term-wise vs Document-wise Partitioning

In one of the earliest publications, Tomasic and Garcia-Molina [147] simulated a distributed multi-disk system used for Boolean queries. Term-wise and document-wise partitioning as we know them were introduced in the work by Jeong and Omiecinski [79]. Similar to Tomasic and Garcia-Molina, Jeong and Omiecinski cover processing of Boolean text queries in a distributed multi-disk system. This time, the system is shared-everything and the skew in the artificial query-term distribution is controlled by an input parameter. Their results show that with a uniform distribution term-wise partitioning performs best, while a skew distribution favors document-wise partitioning. Some of the results indicate that term-wise partitioning may achieve higher throughput at higher multiprogramming levels (given that the skew in distribution is low), while document-wise partitioning gives shorter latency and better load balancing.

The paper by Ribeiro-Neto and Barbosa [127] is the first work that compares term-wise and document-wise partitioning using a TREC document collection, a small number of real queries and the vector space model. According to this work, term-wise partitioning can be considered as an efficient approach to trade-off the number of disk accesses for the network transfer volume. The simulation results show that term-wise partitioning performs better than document-wise for default system properties, but worse when the fast-disk or slow-network scenarios are used.

The paper has three interesting remarks, which explain its results. First, it says that as the number of nodes increases, the number of disk-seeks that have to be performed locally drops. Second, posting list entries that are stored as normalized weight scores and represented as four-byte records without any additional compression make processing more disk-expensive and less CPU-expensive. Third, the model assumes that each node fully processes its sub-query and returns a number of top-results. For term-wise partitioning, the number of partial results returned to the merger node (i.e., the broker) has to be somewhat larger than for document-wise partitioning, but significantly smaller than returning all partially scored documents.

The first evaluation using a real implementation and the Okapi BM25 model was done by MacFarlane et al. [101]. The described system uses one top-node and several leaf-nodes. The top-node works as a ranker for term-wise partitioning and result-merger for document-wise, while the leaf-nodes work as fetchers for term-wise (i.e., no actual processing is done) and as fetchers and rankers for document-wise partitioning. Further, the multiprogramming level is limited to one query at a time. As follows from these assumptions, the results illustrate poor performance, poor load balancing and a potential bottleneck at the top-node when term-wise partitioning is applied.

The work by Badue et al. [12] can be viewed as extension of the work by Ribeiro-Neto and Barbosa [127], this time evaluated on a real system. The results presented in this paper show that term-wise partitioning is more efficient when the average number of terms per query is less than the number of nodes. According to the paper [12], this happens because of an increasing inter-query concurrency ($N/|q|_{avg}$) and decreasing number of disk-seeks per node ($|q|_{avg}/N$). However, a closer look shows that the implementation does not restrict the number of queries processed concurrently and that the total processing time reflects the maximum throughput rather than the average processing latency. Surprisingly, the later work by Badue et al. [13, 14] considers a document-wise partitioned index.

An impressive evaluation has been presented by Moffat et al. [112] (the original work on pipelined query processing). The authors used eight workstations, the GOV2 document collection and a partially synthetic query set. Performance was measured in terms of *normalized throughput*, which is (queries$\times$collection size) / (machines$\times$elapsed time), amount of data read from disk and I/O load, and the number of nodes and collection size have been varied in order to measure scalability of the methods. Their result show that when the number of machines grows proportionally to the collection size, the normalized throughtput decreases only slightly when document-wise partitioning is used, while it falls down when term-wise partitioning is used. It is worth to note that the roles of the nodes were similar to MacFarlane et al., thus the network capacity and processing cost at the query broker were the limiting factors for term-wise partitioning. Some interesting thoughts on the methodology behind, challenges and pitfalls experienced during this evaluation can be found in the two related publications [115, 152].

The paper by Cambazoglu et al. [44] presents a comparison between term-wise and document-wise partitioned indexes using the MPI framework and 32 processing nodes. The experimental results show that, for the given experimental settings, term-wise partitioned indexes give higher throughput, but also longer query latency. The paper illustrates that with an increasing number of processing nodes the number of disk seeks increases when

document-wise partitioning is used, and remains constant when term-wise partitioning is used. The model applied in this work assumes that, for both partitioning methods, each of the processing nodes sends a list of partial document scores (accumulators). Then, these are combined by the broker node in order to produce a top-$k$ result set. Cambazoglu et al. remark that with an increasing index size the broker may become a bottleneck.

Marin et al. [108] have shown that term-wise partitioning achieves superior performance when combined with impact-ordered posting lists, bulk-synchronous processing (BSP) and intersection semantics. However, the query processing approach taken in this and several later publications [105, 106, 107] can be characterized as an iterative retrieval of small fragments of each posting lists and sending these to one of the ranker nodes. Furthermore, retrieval, transfer and ranking are organized in *supersteps* and some of these publications [105] elaborate on a combination of BSP, round-robin ranker assignment and switching between synchronous and asynchronous query processing depending on the query load.

### Hybrid Partitioning

There has been research on combining the advantage of term-wise and document-wise partitioning. Sornil et al. [140] and Xi et al. [155] have presented and evaluated a hybrid partitioning scheme that splits each posting list in chunks and stripes them over several pseudo-randomly chosen nodes. Their results show that this method outperforms both term-wise and document-wise partitioned indexes because of improved I/O and load balancing. However, the results can be explained by the fact that the model used in the experiments performs no actual scoring or ranking, but only a fetch operation. For a general case, this approach is rather impractical, because ID ranges of posting lists stored on each node will be different.

Furthermore, Marin et al. [104] have proposed an approach that allows a document-wise distributed index to be gradually converted into a partial term-wise distributed index at query processing time. Their analytical results show that this approach allows a lower query processing communication cost than term-wise partitioning, while it has a lower indexing communication cost than document-wise.

Finally, Feuerstein et al. have proposed to partition an index in two dimensions, i.e., both by term and by document. The original paper [70] presents a simple analytical evaluation, while a later extension [69] combines 2D partitioning with replication, caching and more advanced query processing methods. Another extension can be found in the work by Marin et al. [106], which combines clustered 2D partitioning, BSP and caching.

### Pipelined Query Processing

Pipelined query processing was introduced by Moffat et al. [112]. The motivation behind this approach is to retain the advantage of term-wise partitioning but to reduce or eliminate the overhead at the broker. The experimental results presented by Moffat et al. show that this method performs significantly better than traditional term-wise partitioning. However,

because of high load-imbalance, it is less efficient and less scalable than document-wise partitioning.

The load balancing issue was addressed in the follow-up paper by Moffat et al. [111] and the final thesis by Webber [151]. In the paper [111], the load balancing is improved by calculating the workload $L_t = Q_t \times B_t$ for each term and assigning posting lists in a fill-smallest fashion. $Q_t$ denotes the frequency of a term in the training query log and $B_t$ denotes the size of the corresponding posting list. Additionally, the authors suggested to multi-replicate some of the posting lists with the highest workloads. The broker node, which decides the query route, can then apply the fill-smallest approach to choose which replica to use. According to the results [111], the modified approach shows a significant improvement above the original one, but it is still less efficient than document-wise partitioning.

Webber's master thesis [151] presents a more detailed description of the original implementation and more recent and extended results. It also evaluates several different strategies for choosing the query route, including the evaluator-based strategy where a node may ask the other nodes to report their load in order to choose the least loaded node as the next step in the route. An extended scalability evaluation with a reduced amount of main memory available for disk-caching shows that pipelined query processing outperforms a document-wise partitioned index in terms of the maximum throughput. However, document-wise partitioning offers shorter latency at low multiprogramming levels. The thesis concludes that the choice between these two methods should therefore rely on whether the latency or the throughput that is the most important.

### Load Balancing

Load balancing and communication cost reduction for term-wise partitioned indexes have been addressed in a number of other publications. In particular, Cambazoglu et al. [42] formulated this task as a hypergraph partitioning problem (however, without consideration of actual queries). Lucchese et al. [99] formulated it as a linear optimization problem solved with a greedy approximation. Zhang et al. [163] presented a solution combining graph partitioning and a greedy heuristic.

Load imbalance can be an issue not only for term-wise partitioned indexes. Badue et al. [13, 14] have studied processing of conjunctive queries over a document-wise partitioned index and reported that the maximum throughput, i.e., the efficiency at higher arrival rates, can be limited because of load imbalance. Furthermore, a more recent paper by Badue et al. [11] shows that the load imbalance across homogeneous index servers is caused by the correlation between the term and query log frequencies and disk caching. Next, a paper by Puppin et al. [126] and several more recent publications address query-based clustering for document-wise partitioning and further collection selection at query processing time. Also, a recent paper by Ma et al. [100] presents a very advanced approach that tries to eliminate the communication overhead in document-wise distributed inverted indexes. Finally, Marin et al. [107] have looked at load balancing of the query traffic. Differently from the others, they addressed scheduling of query processing tasks independent from the actual assignment of the index.

## 2.4 Performance Efficiency and Evaluation

Query processing efficiency can be measured in several different ways. Chowdhury and Pass [52] define *response time* (*latency*), *throughput* and *utilization* as the three main efficiency indicators. The authors suggest that a search engine can be scaled by means of partitioning and replication and present an analytical approach that meets the operational constraints and at the same time minimizes the system's cost. The authors remark that, alternatively, the improvement can be achieved by either buying more expensive hardware, or by improving the software.

In this thesis, we try to improve the performance from a software perspective. For this reason, we may for example fix the system's constructional cost and try to maximize the throughput. This strategy is suitable for batch queries. However, for online queries we have to provide a sufficient throughput and minimize the response time instead. A trade-off between the latency and the throughput can be achieved by switching between intra-query and inter-query concurrency and controlling the multiprogramming level, while caching and pruning improve both of them and additionally reduce the utilization. In presence of result caches we may need to control result freshness (result age), while in presence of pruning optimization we have to ensure result quality. For the latter purpose, we can look at *precision* (number of correct results retrieved to the number of results retrieved) and *recall* (number of correct results retrieved to the number of correct results). When we employ the query degradation mechanism described in Section 2.1.6, we have to minimize the degradation as well. Finally, we remark that our current results overlook the construction and operational costs [27, 78], but we would like to consider them in future.

Performance evaluation can be done with either an analytical model, simulation or actual implementation. An analytical model is the easiest way to predict the performance. However, it either provides a very rough estimation, or quickly grows in the complexity. Additionally, it requires parameter estimation and in some cases it will not show the actual challenges of a real implementation. A real implementation, on the other hand, measures the actual performance. However, it requires a realization of low-level details and an actual execution, which are highly time-consuming. The results may also be prone to the limitations of the implementation and available resources. The simulation model proves a trade-off between an analytical model and an actual implementation as it allows to perform an execution at a certain level of detail and to isolate certain aspects of the system. It provides less realistic results than an actual implementation, but more realistic than an analytical model. As we show in the next section, we apply all three methods dependent on the problem, requirements for realistic results and available resources. In particular, Papers A.II–A.IV, A.VI, B.I–B.III use a real implementation, while Papers A.I and C.II apply simulation and Paper C.I combines simulation and an analytical model.

# Chapter 3

# Research Summary

*"Life is a series of experiences, each of which makes us bigger, even though it is hard to realize this. For the world was built to develop character, and we must learn that the setbacks and grieves which we endure help us in our marching onward."*

– Henry Ford

This chapter describes the research behind this thesis. Sections 3.1 and 3.2 briefly overview the research process and the publications. A detailed overview of the included papers can be found in Sections 3.3. A similar overview of the secondary publications can be found in Section 3.4. A description of the frameworks and data is given in Section 3.5. Finally, Section 3.6 evaluates our contributions and Section 3.7 concludes the thesis.

## 3.1 Formalities

The work described in this thesis was completed during a four-year PhD program, which was supported by the iAd Centre, financed by the Research Council of Norway and NTNU and included 25% duty work. The duty work was carried out in the courses TDT4225 Management of Very Large Data Volumes and TDT4145 Data Modeling, Databases and Database Management Systems throughout the whole period (eight semesters). Further, I took the following courses as a part of the PhD program: IT8802 Advanced Information Retrieval, DT8116 Web Data Mining, DT8105 Computer Architecture II, TDT4225 Management of Very Large Data Volumes, DT8108 Topics in Information Science. Four of the courses included a final exam, as well as obligatory practical assignments throughout the course. Additionally, I took TDT4215 Web Intelligence, which was highly useful as an introduction to search engine implementation. Moreover, the work done in the periods 15.09.11–15.12.11 and 22.01.12–17.02.12 was carried out at Yahoo! Research, Barcelona, while the rest of the work (03.08.08–04.09.12) was carried out at IDI, NTNU, Trondheim, with exception for conference and workshop attendance and vacations.

Table 3.1: A complete overview of the papers.

| ID | Title | Ref. | Incl. |
|---|---|---|---|
| A.I | Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes. | [83] | – |
| A.II | A Combined Semi-Pipelined Query Processing Architecture for Distributed Full-Text Retrieval. | [84] | ✓ |
| A.III | Improving the Performance of Pipelined Query Processing with Skipping. | [86] | ✓ |
| A.IV | Intra-Query Concurrent Pipelined Processing for Distributed Full-Text Retrieval. | [87] | ✓ |
| A.V | Scalable Search Platform: Improving Pipelined Query Processing for Distributed Full-Text Retrieval. | [82] | – |
| A.VI | A Term-Based Inverted Index Partitioning Model for Efficient Query Processing. | – | – |
| B.I | Efficient Compressed Inverted Index Skipping for Disjunctive Text-Queries. | [85] | ✓ |
| B.II | Efficient Processing of Top-k Spatial Keyword Queries. | [132] | – |
| B.III | Improving Dynamic Index Pruning via Linear Programming. | – | ✓ |
| C.I | Modeling Static Caching in Web Search Engines. | [20] | ✓ |
| C.II | Prefetching Query Results and its Impact on Search Engines. | [88] | ✓ |

## 3.2 Publications and Research Process

As shown in Table 3.1, the publications that I have contributed to can be divided in three categories:

Category A addresses partitioned query processing. This work started during my master thesis [81] and led to several publications on pipelined query processing. In particular, Paper A.II presents semi-pipelined query processing, Paper A.III improves pipelined query processing with skipping and pruning, and Paper A.IV addresses intra-query concurrent pipelined processing. Additionally, this category includes a paper summarizing the results of my master thesis (Paper A.I, not included), the ongoing work on hypergraph partitioning methods (Paper A.VI, not included), and the PhD symposium paper appeared at WWW 2012 (Paper A.V, not included).

Category B mainly addresses skipping and pruning. This work started as a part of the work on pipelined query processing with skipping and led to several other, non-distributed optimizations. Paper B.I addresses compression-efficient skipping and pruning methods (including Max-Score), while Paper B.III presents a further linear programming optimization of Max-Score. Additionally, this category includes a paper on spatio-textual query processing (Paper B.II, not included).

Category C addresses caching and contains the work on static cache modeling presented in Paper C.I and result prefetching presented in Paper C.II.

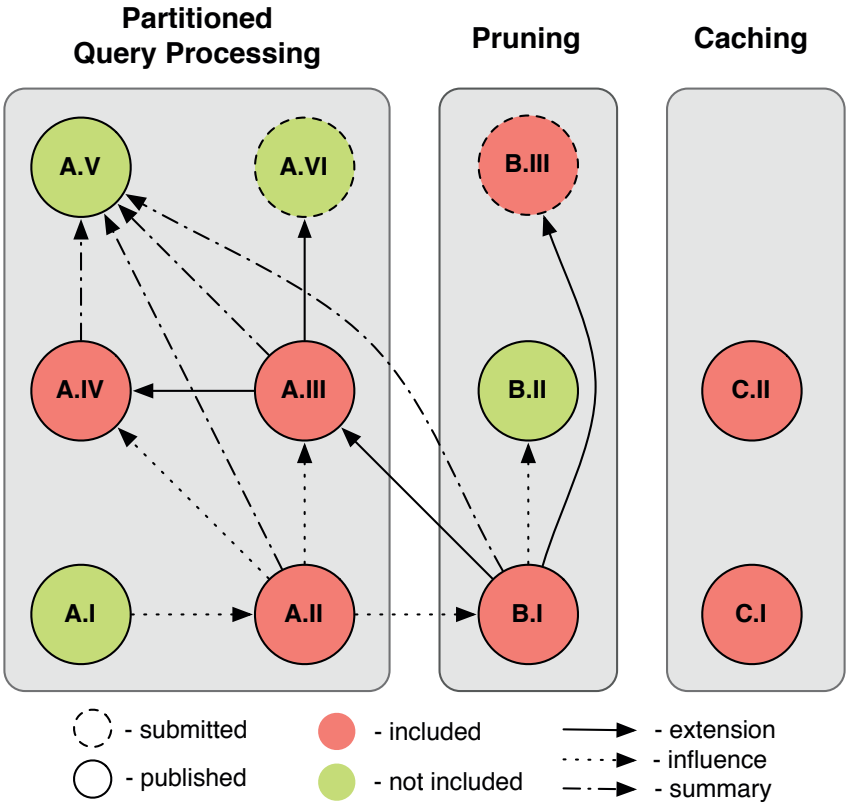



Figure 3.1: A logical overview of the papers.

The relationships between the papers are illustrated in Figure 3.1. The observations made in Paper A.I led to the work on Paper A.II, which in its turn led to Papers A.III and A.IV. Paper B.I was written as a part of the work on Paper A.III, which tries an alternative to the approach presented in Paper A.II. Papers A.IV and A.VI extend one of the methods presented in Paper A.III, while Paper A.V summarizes the work presented in Papers A.II, A.III, A.IV and B.I as an attempt to resolve the challenges of pipelined query processing. Further, Paper B.III extends one of the methods from Paper B.I, while Paper B.II spins into a completely different research field (as a consequence of an attempt to extend Paper B.I). Finally, Papers C.I and C.II are two independent contributions.

## 3.3 Included Papers

This section describes the papers that are included in the thesis. The actual papers can be found in the Appendix.

### 3.3.1 Paper A.II

**A Combined Semi-Pipelined Query Processing Architecture for Distributed Full-Text Retrieval**

Simon Jonassen and Svein Erik Bratsberg

*Proceedings of the 11th International Conference on Web Information Systems Engineering (WISE), pages 587–601, Springer 2010.*

**Abstract:** Term-partitioning is an efficient way to distribute a large inverted index. Two fundamentally different query processing approaches are pipelined and non-pipelined. While the pipelined approach provides higher query throughput, the non-pipelined approach provides shorter query latency. In this work we propose a third alternative, combining non-pipelined inverted index access, heuristic decision between pipelined and non-pipelined query execution and an improved query routing strategy. From our results, the method combines the advantages of both approaches and provides high throughput and short query latency. Our method increases the throughput by up to 26% compared to the non-pipelined approach and reduces the latency by up to 32% compared to the pipelined.

**Research process:** Our original plan was to look at 2D partitioning. The hypothesis was that partitioning by both documents and terms will counterbalance the number of disk-seeks and the reading, processing and network latencies. In order to evaluate this idea, I extended Terrier [118] to a distributed system and ran it on the nine-node cluster offered by the HPC Group. However, because of an insignificant improvement and after discovering the publications by Feuerstein et al. [69, 70] this work has been abandoned.

While trying to improve the performance of the term-wise partitioned index, I implemented Lester's space-limited pruning method[1] [94] and pipelined query processing [112]. The experiments with these methods led to two interesting observations. First, even with a considerably large accumulator set target size ($L = 400\ 000$, given $k = 1\ 000$), Lester's pruning method was several times faster than a full evaluation. Second, pipelined query processing resulted in a higher latency at low multiprogramming levels, but also a higher maximum throughput at higher multiprogramming levels. The latter observation led to the semi-pipelined approach presented in this paper. The main motivation for this work was to unite the advantage of pipelined and non-pipelined query processing.

**Roles of the authors:** I came up with the ideas, did the implementation, experiments and writing. Prof. Bratsberg participated in technical discussions and writing process, and helped with constructive comments and feedback.

[1] Technically, our implementation is based on the `thresh_decode` method found in `okapi_k3.c`, Zettair Search Engine (`http://www.seg.rmit.edu.au/zettair/`), v.0.9.3

**Retrospective view:** While having received the Best Paper Award of the conference, the paper has several shortcomings. The most important of these are the index compression and posting list access methods and a large memory footprint (these issues motivated the work described in Papers A.III, A.IV and B.I). Furthermore, in order to reduce the number of experiments the accumulator set target ($L$) has been fixed at $400\ 000$. We admit that a different choice of $L$ may alter the relationship between the two baselines, as well as the proposed solution. Finally, while the paper states that term-wise partitioning is an efficient way to distribute an index, the results in Appendix H show that its performance may be inferior to that attained by document-wise and 2D partitioning in most of the cases.

### 3.3.2 Paper A.III

**Improving the Performance of Pipelined Query Processing with Skipping**
Simon Jonassen and Svein Erik Bratsberg
*Proceedings of the 13th International Conference on Web Information Systems Engineering (WISE), pages 1–15, Springer 2012.*

**Abstract:** Web search engines need to provide high throughput and short query latency. Recent results show that pipelined query processing over a term-wise partitioned inverted index may have superior throughput. However, the query processing latency and scalability with respect to the collections size are the main challenges associated with this method. In this paper, we evaluate the effect of inverted index skipping on the performance of pipelined query processing. Further, we introduce a novel idea of using Max-Score pruning within pipelined query processing and a new term assignment heuristic, partitioning by Max-Score. Our current results indicate a significant improvement over the state-of-the-art approach and lead to several further optimizations, which include dynamic load balancing, intra-query concurrent processing and a hybrid combination between pipelined and non-pipelined execution.

**Research process:** Based on the observations made upon writing Paper A.II, I wanted to extend pipelined query processing with an efficient combination of DAAT sub-query processing, safe pruning and skipping. The hypothesis was that this will reduce the memory footprint, I/O, decompression cost and the amount of non-useful processing. Additionally, it would improve the retrieval performance and allow intra-query concurrent processing. Inspired by discussions with Ola Natvig, who was at that time working on his master thesis [116], I wanted also to take a look at the PFoR compression methods (see Sec. 2.1.3), which were remarkably efficient according to his results.

The work was divided in three papers. Paper B.I addresses query processing on a single node, Paper A.III addresses skipping with pipelined query processing, and Paper A.IV addresses intra-query concurrency. The work on Paper A.III was originally done during the spring 2011. Because of initial publication problems, the work was redone one year later.

**Roles of the authors:** Same as Paper A.II.

**Retrospective view:** The approach presented in the paper is orthogonal to the one in Paper A.II. The second (current) version of the paper has been completely rewritten with respect to the comments received from the reviewers. The experiments were redone in accordance to the framework and methodology improvements achieved upon working on Papers A.III and A.VI.

The most important limitations of this work are as follows. First, the performance linearity evaluation was done by scaling the document collection down, which means that it cannot predict what happens when a larger document collection or a larger number of nodes is used. Second, the paper lacks a comparison with document-wise partitioning as another baseline. A more recent fixed-scale comparison between $\mathrm{MSD}^{\star}_{\mathrm{S}}$ and $\mathrm{AND}_{\mathrm{S}}$ and a similar implementation of document-wise partitioning with Max-Score pruning can be found in Appendix H (Figure H.4 and the related part of discussion). Assuming that load balancing and communication overhead associated with pipelined processing will be even more critical on a larger scale, document-wise partitioning retains the advantage.

### 3.3.3 Paper A.IV

**Intra-Query Concurrent Pipelined Processing for Distributed Full-Text Retrieval**
Simon Jonassen and Svein Erik Bratsberg
*Proceedings of the 34th European Conference on Information Retrieval (ECIR), pages 413–425, Springer 2012.*

**Abstract:** Pipelined query processing over a term-wise distributed inverted index has superior throughput at high query multiprogramming levels. However, due to long query latencies this approach is inefficient at lower levels. In this paper we explore two types of intra-query parallelism within the pipelined approach, parallel execution of a query on different nodes and concurrent execution on the same node. According to the experimental results, our approach reaches the throughput of the state-of-the-art method at about half of the latency. On the single query case the observed latency improvement is up to 2.6 times.

**Research process:** Intra-query concurrency and query processing on multi-core architectures have been one of my topics of interest since the beginning. It was also the topic for my term project in Computer Architecture II written in the spring 2009, where I discussed several techniques and possibilities for a further research. Despite receiving a highly positive feedback from Prof. Lasse Natvig and an external reviewer, I did not follow this direction for single-node processing, as I discovered several highly related publications [63, 64, 145] and an extended version of the work by Tatikonda et al. [145], which was somehow cached by Google (their final and significantly altered version was presented at SIGIR 2011 [144]). Therefore, I concluded that working in this direction was very risky. Some of the ideas described in the term project were however reused in this paper. The questions I wanted to address in this work were how to extend pipelined processing with intra-query concurrency, whether it would improve the performance at lower multiprogramming levels and how big the improvement would be.

**Roles of the authors:** Same as Paper A.II.

**Retrospective view:** We are very grateful for the respectful and constructive questions and comments received from Dr. Ismail Sengör Altingövde, Prof. Arjen P. De Vries and Dr. Fabrizio Silvestri during our presentation at ECIR. In the following, we briefly summarize some of these comments. First, one of the limitations of this work (as well as Papers A.II, A.III, A.VI and B.I–B.III) is the presence of OS disk cache. There are several ways to go around this problem, however it can be questioned whether it will give a 'more realistic' performance or not. Instead, we reset the disk cache and perform a warm-up before each run. Second, because of the implementation in Java, the amount of low-level thread control was limited. An implementation in, e.g., C or C++ would allow to control processor affinity, fine-tune the actual number of threads, etc., which could lead to an even better performance improvement. Finally, the methods described in the paper have many variables and options to be tuned, but we tried to describe an approach that solves an existing problem, rather than to tweak the actual performance.

The most important limitations of this work are as follows. First, the paper builds on the assumption that pipelined query processing over a term-wise partitioned distributed index has superior maximum throughput. This might be correct on the given system/implementation with the given query log, but the results presented in Appendix H indicate that in other cases or on another scale the maximum throughput of term-wise partitioning may be inferior to that attained by document-wise partitioning. Second, the techniques described in this paper aim to improve the query latency when the query load is low or moderate. Because of the performance degradation due to parallelization overhead, they provide no benefit at high query loads. Consequently, these methods are interesting only when query latency is important and assuming that pipelined query processing over a term-wise distributed index is used.

### 3.3.4 Paper B.I

**Efficient Compressed Inverted Index Skipping for Disjunctive Text-Queries**
Simon Jonassen and Svein Erik Bratsberg
*Proceedings of the 33rd European Conference on Information Retrieval (ECIR), pages 530–542, Springer 2011.*

**Abstract:** In this paper we look at a combination of bulk-compression, partial query processing and skipping for document-ordered inverted indexes. We propose a new inverted index organization, and provide an updated version of the MaxScore method by Turtle and Flood and a skipping-adapted version of the space-limited adaptive pruning method by Lester et al. Both our methods significantly reduce the number of processed elements and reduce the average query latency by more than three times. Our experiments with a real implementation and a large document collection are valuable for a further research within inverted index skipping and query processing optimizations.

**Research process:** The first phase of the work addressed in Papers B.I, A.III and A.IV (see the description of Paper A.III) included design and implementation of an efficient

DAAT query processing method for a non-distributed index. The motivation was to provide safe and yet efficient pruning, efficiently skip in long posting lists, maintain a small memory footprint and optimize the structure for compact representation, compression and buffering. The query processing approach and the skipping structure have evolved together in a design process augmented with implementation. Unfortunately, after finishing the experiments, I discovered that the approach was nothing else than DAAT Max-Score. I knew that a part of the idea was similar to the Max-Score heuristic described by Turtle and Flood [150]. However, while writing the related work section, I found that the description of the method by Strohman et al. [143] (i.e., the last three paragraphs of the related work section) was the missing link between Max-Score and my description. This led to the corresponding changes in the paper. In order to add more contribution, the paper was extended with a skipping version of the pruning method by Lester et al. [94].

**Roles of the authors:** Same as Paper A.II.

**Retrospective view:** The biggest mishap of this work was reinventing Max-Score. Next, the paper may have been better if it focused only at Max-Score, but covered both DAAT and TAAT versions. This would also improve the contribution and the content of Paper A.III. Further, the evaluation part of the paper could be improved by providing results for varied query length and collection size. Finally, the description of the work by Broder et al. [37] given in the paper is incorrect. For a correct description we refer to Section 2.2.2 (I express my gratitude to Dr. Craig Macdonald for pointing this out).

Right after finishing this paper, we were working on an extension of the skipping structure that could skip not only by document ID, but also, for example, by minimum required impact, context ID or geographical boundaries. The main motivation was to increase the distance of actual skips and therefore to reduce I/O and the decompression and processing costs. However, after doing some experiments, we postponed all work in this direction and went back to distributed query processing, which we thought was more promising. In fact, query processing methods presented later by Ding et al. [65] (Block-Max WAND) and Christoforaki et al. [53] (Space-Filling Curve methods) fit perfectly on top of this extension.

### 3.3.5 Paper B.III

**Improving Dynamic Index Pruning via Linear Programming**
Simon Jonassen and B. Barla Cambazoglu
*under submission / in progress*

**Abstract:** Dynamic index pruning techniques are commonly used to speed up query processing in web search engines. In this work, we propose a linear programming technique that can further improve the performance of the state-of-the-art dynamic index pruning techniques. The experiments we conducted show that the proposed technique achieves reduction in terms of the disk access, index decompression, and scoring costs compared to the well-known Max-Score technique.

**Research process:** This work started during my internship at Yahoo! Research. Dr. Cambazoglu suggested to look at the linear programming (LP) approach for upper-bound score estimation described in one of his papers [47] and see whether a similar approach could improve the efficiency of WAND and Max-Score on conjunctive (AND) keyword queries. While working on this idea, we reduced the scope to Max-Score only (because it does not require to reorder posting lists after each iteration), but covered disjunctive (OR) queries as well. The implementation and experiments were done in December 2011 and writing took place in March 2012.

**Roles of the authors:** I came up with the algorithms, did the implementation and the experiments, and wrote the initial version of the paper. Dr. Cambazoglu came up with the general idea, provided the data and the LP solver, offered valuable feedback during the process and helped to improve the focus and the presentation of the paper.

**Retrospective view:** Our results have shown a significant improvement in pruning efficiency, while the actual latency improvement was insignificant (because of a relatively high cost of the LP computation and surprisingly fast query evaluation). However, as the LP computation itself depends only on the number of terms in each query (or more exactly, the number of known term-pair scores), we believe that this technique will be much more beneficial for a larger index. A similar approach can also be used to improve the performance of Max-Score pipelined query processing described in Papers A.III and A.IV. Finally, there is a possibility for reducing the cost of the LP part by segmenting each query into several strong components (of term-pairs) and computing LP bounds within each component independent from the others, and/or by caching partial LP scores.

### 3.3.6 Paper C.I

**Modeling Static Caching in Web Search Engines**
Ricardo Baeza-Yates and Simon Jonassen
*Proceedings of the 34th European Conference on Information Retrieval (ECIR), pages 436–446, Springer 2012.*

**Abstract:** In this paper we model a two-level cache of a Web search engine, such that given memory resources, we find the optimal split fraction to allocate for each cache, results and index. The final result is very simple and implies to compute just five parameters that depend on the input data and the performance of the search engine. The model is validated through extensive experimental results and is motivated on capacity planning and the overall optimization of the search architecture.

**Research process:** This work was done during my internship at Yahoo! Research. Prof. Baeza-Yates had the idea prior to my arrival. However, the model needed an estimation of several parameters and an evaluation, which were accomplished by my contribution.

**Roles of the authors:** Prof. Baeza-Yates came up with the idea and wrote the paper. I did the implementation, experiments, and contributed to writing with experimental details, figures and feedback. We also participated in technical discussions throughout the process.

**Retrospective view:** A generalization of the modeling approach to other types of cache is an interesting direction for further work.

### 3.3.7 Paper C.II

**Prefetching Query Results and its Impact on Search Engines**
Simon Jonassen, B. Barla Cambazoglu and Fabrizio Silvestri
*Proceeding of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 631–640, ACM 2012.*

**Abstract:** We investigate the impact of query result prefetching on the efficiency and effectiveness of web search engines. We propose offline and online strategies for selecting and ordering queries whose results are to be prefetched. The offline strategies rely on query log analysis and the queries are selected from the queries issued on the previous day. The online strategies select the queries from the result cache, relying on a machine learning model that estimates the arrival times of queries. We carefully evaluate the proposed prefetching techniques via simulation on a query log obtained from a commercial search engine. We show that our strategies are able to improve various performance metrics, including the hit rate, query response time, result freshness, and query degradation rate, relative to a state-of-the-art baseline.

**Research process:** The work was done during my internship at Yahoo! Research. Since my introduction to Dr. Cambazoglu, we were looking for an important, open problem that could lead to a full paper in one of the top-level conferences. During this period, Dr. Cambazoglu supplied me with interesting papers and we discussed possible extensions. At some point, we discussed the batch-processing paper by Ding et al. [62] and Dr. Cambazoglu suggested to combine this work with the recent developments in result caching, namely the time-based invalidation (see Sec. 2.2.1).

Dr. Silvestri joined us in late November and we started to discuss possible contributions. During these discussions, we defined the scope of the paper and came up with the architecture. We also decided to submit the paper to SIGIR 2012. As my internship was ending in December, we decided that I should come back and stay for 3 more weeks, right before the deadline. Most of the writing of the introduction, related work and the main parts of the paper, and the initial implementation were done over the Christmas, while the rest (including the specific methods and the actual experiments) came together after my return to Barcelona.

**Roles of the authors:** I contributed to the main idea, came up with the online methods, did the implementation, designed and conducted the experiments, and wrote the initial version of the experimental setup and results sections. Dr. Cambazoglu contributed to the main idea, came up with the offline methods and the machine learned prediction model, and wrote the initial version of the preliminaries and related work sections. Dr. Silvestri contributed to the main idea, wrote the initial version of the abstract, introduction and conclusions and further work sections. Finally, each of the authors participated in the discussions throughout the process, provided valuable feedback and ideas and contributed to writing of the final version.

**Retrospective view:** In the beginning of this work, we were mostly concerned about the architecture. However, it turned out that prediction of the next query arrival is the main challenge. Our current results suffer the most from a relatively short learning stage. Consequently, the model is unable to capture the queries that reappear with an interval larger than one day. Therefore, a greater improvement can be achieved by elaborating on the prediction model. Additionally, our approach was to utilize the backend as much as possible, while it might be possible to vary the load in order to minimize the electricity bill [89].

## 3.4 Other Papers

This section describes the papers that I have contributed to as a part of this research, but that are not included in this thesis.

### 3.4.1 Paper A.I

**Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes**

Simon Jonassen and Svein Erik Bratsberg

*Proceedings of the 22nd Norwegian Informatics Conference (NIK), pages 143–151, Tapir Akademisk Forlag 2009.*

**Abstract:** This paper presents an evaluation of three partitioning methods for distributed inverted indexes: local, global and hybrid indexing, combined with two generalized query models: conjunctive query model (AND) and disjunctive query model (OR). We performed simulations of various settings using a dictionary dump of the TREC GOV2 document collection and a subset of the Terabyte Track 05 query log. Our results indicate that, in situations when a conjunctive query model is used in combination with a high level of concurrency, the best performance and scalability are provided by global indexing with pipelined query execution. For other situations, local indexing is a more advantageous method in terms of average query throughput, query wait time, system load and load imbalance.

**Acknowledgment:** This paper extends my master thesis [81] and therefore is not included in the thesis.

**Research process:** The paper was written during the first year of my PhD study. It briefly summarizes the main results of my master thesis, as well as the related work. The thesis compared the query processing efficiency of different index partitioning methods using a simulation framework and was supervised by Prof. Bratsberg and Dr. Torbjørnsen.

**Roles of the authors:** Same as Paper A.II.

**Retrospective view:** The results of this work motivated us to look at 2D partitioning and pipelined query processing (see Appendix H). The simulation framework was written in

Java using the Desmo-J[2] API and included elements of micro-benchmarking. Because of an overly complicated and computation-intensive simulation model, each run was shorten to just 50 simulated seconds. Nevertheless, the relative performance of different scenarios captured in this work agrees with our later observations.

### 3.4.2 Paper A.V

**Scalable Search Platform: Improving Pipelined Query Processing for Distributed Full-Text Retrieval (PhD Symposium)**

Simon Jonassen

*(Advised by Prof. Svein Erik Bratsberg and Adj. Assoc. Prof. Øystein Torbjørnsen)*

*Proceedings of the 21st International World Wide Web Conference (WWW, Companion Volume), pages 145–150, ACM 2012.*

**Abstract:** In theory, term-wise partitioned indexes may provide higher throughput than document-wise partitioned. In practice, term-wise partitioning shows lacking scalability with increasing collection size and intra-query parallelism, which leads to long query latency and poor performance at low query loads. In our work, we have developed several techniques to deal with these problems. Our current results show a significant improvement over the state-of-the-art approach on a small distributed IR system, and our next objective is to evaluate the scalability of the improved approach on a large system. In this paper, we describe the relation between our work and the problem of scalability, summarize the results, limitations and challenges of our current work, and outline directions for further research.

**Acknowledgment:** This paper is redundant to the content of Chapter 2 and Papers A.II–A.IV and B.I and therefore is not included in the thesis.

**Research process and retrospective view:** My PhD topic was originally given by the iAd with the following description:

> "*The PhD student will focus on the research activity in the area of scalable search platforms. The research challenge here is to develop methods to scale with respect to search power and data volume to be indexed, and at the same time give good performance and minimizing human management during scaling or churn in the hardware resources.*"

The doctoral consortium paper united Papers A.II–A.IV and B.I as an attempt to resolve the limitations of pipelined query processing and explained the connection to the research topic. When the paper was submitted, in August 2011, my further plans included a scalability evaluation. However, with respect to my later contributions, the focus of the thesis has changed from scalability to efficiency, and the granularity of the addressed system has increased from a cluster to a distributed search engine. A further discussion of this topic can be found in Section 3.6 and Appendix H.

[2] `http://desmoj.sourceforge.net/`

### 3.4.3 Paper A.VI

**A Term-Based Inverted Index Partitioning Model for Efficient Query Processing**
B. Barla Cambazoglu, Enver Kayaaslan, Simon Jonassen and Cevdet Aykanat
*under submission / in progress*

**Abstract:** In a shared-nothing, distributed text retrieval system, queries are processed over an inverted index that is partitioned among a number of index servers. In practice, the index is either document-based or term-based partitioned. This choice is made depending on the properties of the underlying hardware infrastructure, query traffic distribution, and some performance and availability constraints. In query processing on retrieval systems that adopt a term-based index partitioning strategy, the high communication overhead due to the transfer of large amounts of data from the index servers forms a major performance bottleneck, deteriorating the scalability of the entire distributed retrieval system. In this work, to alleviate this problem, we propose a novel combinatorial model that tries to assign concurrently accessed index entries to the same index servers, based on the inverted index access patterns extracted from the past query logs. The model aims to minimize the communication overhead that will be incurred by future queries while maintaining the computational load balance among the index servers. We evaluate the performance of the proposed model through extensive experiments using a real-life text collection and a search query sample. Our results show that considerable performance gains can be achieved relative to the term-based index partitioning strategies previously proposed in literature. In most cases, however, the performance remains inferior to that attained by document-based partitioning.

**Acknowledgment:** This paper will be included in Enver Kayaaslan's PhD thesis and therefore is not included in mine.

**Research process:** This work started during my internship at Yahoo! Research. Because I had a partitioned query processing system running on a real cluster, Dr. Cambazoglu asked me to do the experiments for an extension of his earlier work [42]. During this process, we decided to combine pipelined query processing with Max-Score pruning described in Paper A.III and the methods described in the original paper draft. The work has been carried out in November (implementation and experiments) and December 2011 (writing), and the paper was submitted in January 2012. Despite positive improvements, we were asked to extend the evaluation with additional baselines including document-wise partitioning. The paper has been extended as requested, but remains under submission.

**Roles of the authors:** Dr. Cambazoglu, Kayaaslan and Prof. Aykanat came up with the main idea and wrote an earlier version of the paper. I adapted the idea to pipelined query processing and Max-Score pruning, did the implementation, designed and conducted the experiments and participated in writing of the evaluation section. Kayaaslan performed the query log analysis as well as hypergraph partitioning, and contributed to writing. Dr. Cambazoglu summarized the results and wrote the final version of the paper. Prof. Aykanat contributed to writing and provided helpful suggestions, comments and feedback throughout the process.

### 3.4.4 Paper B.II

**Efficient Processing of Top-k Spatial Keyword Queries**

João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen and Kjetil Nørvåg

*Proceedings of the 12th International Symposium on Spatial and Temporal Databases (SSTD), pages 205–222, Springer 2011.*

**Abstract:** Given a spatial location and a set of keywords, a top-$k$ spatial keyword query returns the $k$ best spatio-textual objects ranked according to their proximity to the query location and relevance to the query keywords. There are many applications handling huge amounts of geotagged data, such as Twitter and Flickr, that can benefit from this query. Unfortunately, the state-of-the-art approaches require non-negligible processing cost that incurs in long response time. In this paper, we propose a novel index to improve the performance of top-$k$ spatial keyword queries named *Spatial Inverted Index* (S2I). Our index maps each distinct term to a set of objects containing the term. The objects are stored differently according to the document frequency of the term and can be retrieved efficiently in decreasing order of keyword relevance and spatial proximity. Moreover, we present algorithms that exploit S2I to process top-$k$ spatial keyword queries efficiently. Finally, we show through extensive experiments that our approach outperforms the state-of-the-art approaches in terms of update and query cost.

**Acknowledgment:** This paper was included in Dr. João B. Rocha-Junior's PhD thesis and therefore is not included in mine.

**Research process:** After finishing Paper B.I, among other things, I have considered an extension of the skipping structure with minimum bounding rectangles (MBRs) over the geographical coordinates of the underlying documents. I expected this to be an interesting direction. However, I was not sure how to reorder documents in order to optimize pruning and was afraid that all MBRs would be more or less the same. I told my ideas to Rocha-Junior, a fellow PhD student who was at that time also looking at spatio-textual query processing. One of my suggestions was that building a large IR tree for the whole collection might be inefficient, and that posting lists may provide a more compact representation and more efficient query processing. The suggestion that came from Rocha-Junior was therefore to build a separate tree (or just a block) for each term. This discussion led to the paper written by Dr. Rocha-Junior, where I contributed as a co-author.

**Roles of the authors:** Dr. Rocha-Junior came up with the ideas, wrote the paper, did most of the implementation, and conducted the experiments. Gkorgkas participated in technical discussions, implemented a part of the baseline approach, created the synthetic datasets, and provided feedback during the writing phase. I was part of the initial idea suggesting to use multiple trees, participated in technical discussions, developed the parser for the Flickr and Twitter datasets, and gave feedback during the writing. Prof. Nørvåg contributed to writing and provided constructive and helpful feedback throughout the process.

**Retrospective view:** Similar to most of the publications on spatio-textual query processing, the paper takes a DB approach to the problem. Here we find synthetic data and trees. However, it integrates parts of an IR point of view, where we find references to Zipf's Law

and use of data from Twitter and Flickr. In contrast to this, there are several works [50, 53] that represent a pure IR approach, where posting lists are augmented with geographical information.

## 3.5 Frameworks and Data

During this research, besides the theoretical aspects, a great effort was put in practical implementation and development of evaluation methodology. The work behind the thesis included development of several query processing and simulation frameworks and experiments with several document collections and query logs, which are briefly described in Sections 3.5.1 and 3.5.2.

### 3.5.1 Frameworks

This section describes the query processing frameworks and simulators that were developed as a part of this research. The framework behind Paper B.II developed by Dr. João B. Rocha-Junior and Orestis Gkorgkas and the simulator behind Paper A.I are not included in this description.

#### SolBrille

SolBrille [73] is a small search engine written together with a group of friends (Arne Bergene Fossaa, Ola Natvig and Jan Maximilian W. Kristiansen), as a term project in Web Intelligence. Although this code (except from the buffer manager written by Natvig) was not used in any of the papers, the implementation was an exciting process, which extended my knowledge and gave me a valuable experience and inspiration for the later work.

SolBrille implements both indexing and query processing, it also supports index updates and provides a Web interface. The query processing approach is much more complicated than in my later work. It involves a pipeline of *components*, each of which allows to pull candidates and in its turn may include another component as a source. As illustrated in Figure 3.2, the pipeline has three basic components. The *matcher* joins the postings based on the optional AND, OR and NAND modifiers of the query terms. The *score combiner* evaluates the candidates using a set of scorers, each implementing a particular relevance model, and calculates a weighted average. Finally, a *filter* removes the candidates mismatching specific constraints, such as phrases or score-specific constraints. The candidates that come out of the pipeline are then inserted into a heap, sorted and presented to the user. For the latter purpose, SolBrille involves a post-processing stage, where it generates snippets and does suffix-tree clustering [76] of the results. Figure 3.3 shows SolBrille in action. Further information can be found in the project report [73] and the code is available online[3].

[3] `http://code.google.com/p/solbrille/`

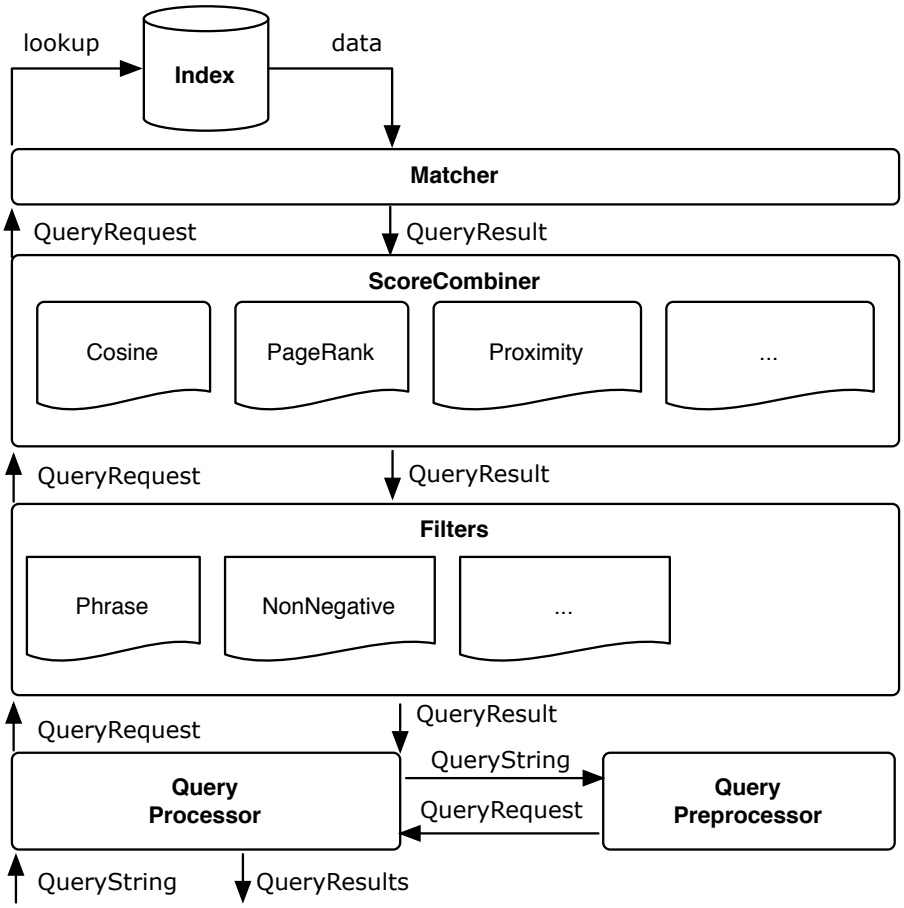



Figure 3.2: Query processing in SolBrille [73].

### Dogsled (Terrier)

The initial plan for the PhD work included development of a distributed query processing framework. The idea was to adapt or extend one of the existing search engines. For this reason, I studied several open-source search engines including Zettair[4], MG4J[5] [32], Terrier[6] [118], Lucene[7] and Solr[8]. Because of its simple and elegant code structure and being widely used by the research community, the choice was to use Terrier. Therefore, I extended Terrier to a distributed search engine named Dogsled. The implementation included writing a communication manager, modifying the code of Terrier by both simplifying the query processing approach and making it distributed, and even writing a distributed indexer. The resulting framework supports both term-wise, document-wise and 2D partitioning and is deployable through a shell script. This framework was used in Paper A.II and some of the experiments described in Appendix H. The code is available on request.

[4] http://www.seg.rmit.edu.au/zettair/
[5] http://mg4j.dsi.unimi.it/
[6] http://terrier.org/
[7] http://lucene.apache.org
[8] http://lucene.apache.org/solr/

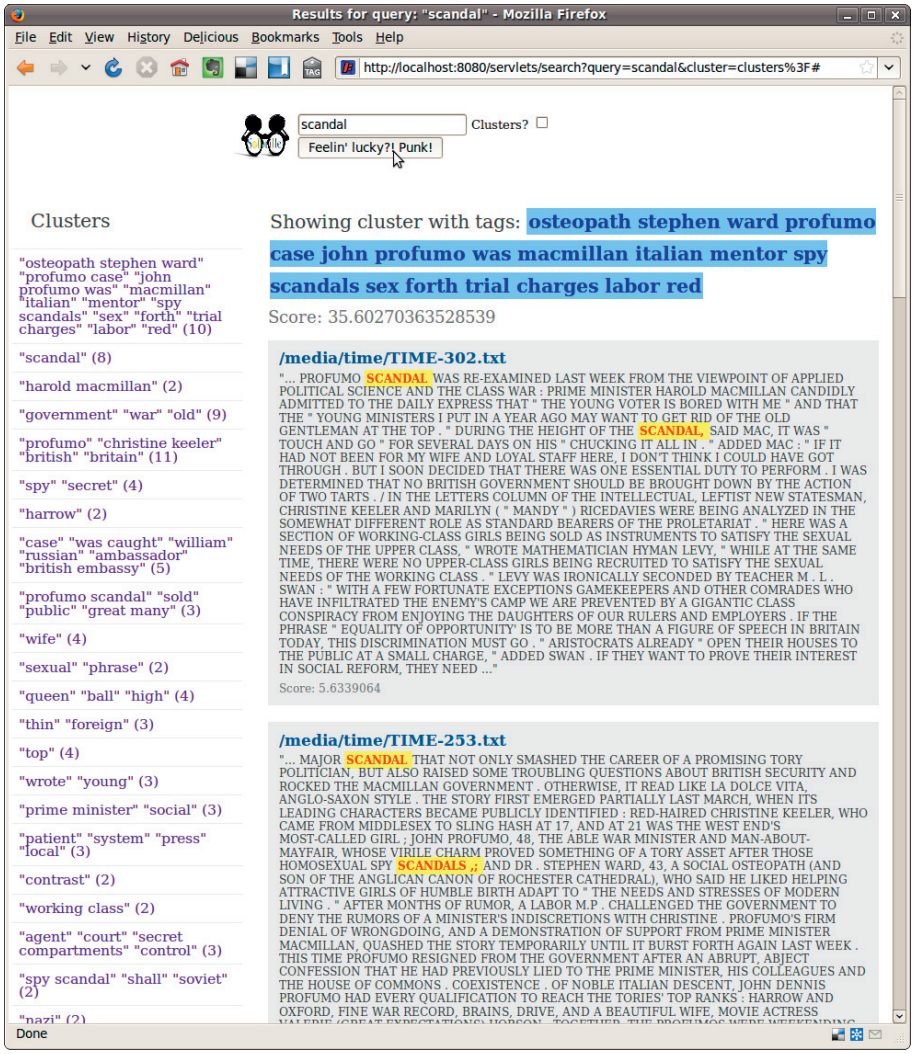



Figure 3.3: SolBrille in action.

### Laika

Along with writing Paper B.I, I started on writing a new framework called Laika. During this period, I saved a lot of time by reusing the buffer manager of SolBrille, porting Natvig's implementation of VByte, Simple-9 and NewPFoR into Java, and writing an index converter for a Terrier-built index rather than implementing indexing myself.

The logical index structure in Laika is equivalent to the one described in Section 2.1. In the actual implementation, the main index structures such as the document dictionary and the term lexicon are very similar to the implementation in Terrier, while the inverted index itself corresponds to the description in Paper B.I.

As a part of the work described in Paper A.III, Laika was extended to a distributed framework. Reusing the architecture and best-practices learned from my work on Dogsled,

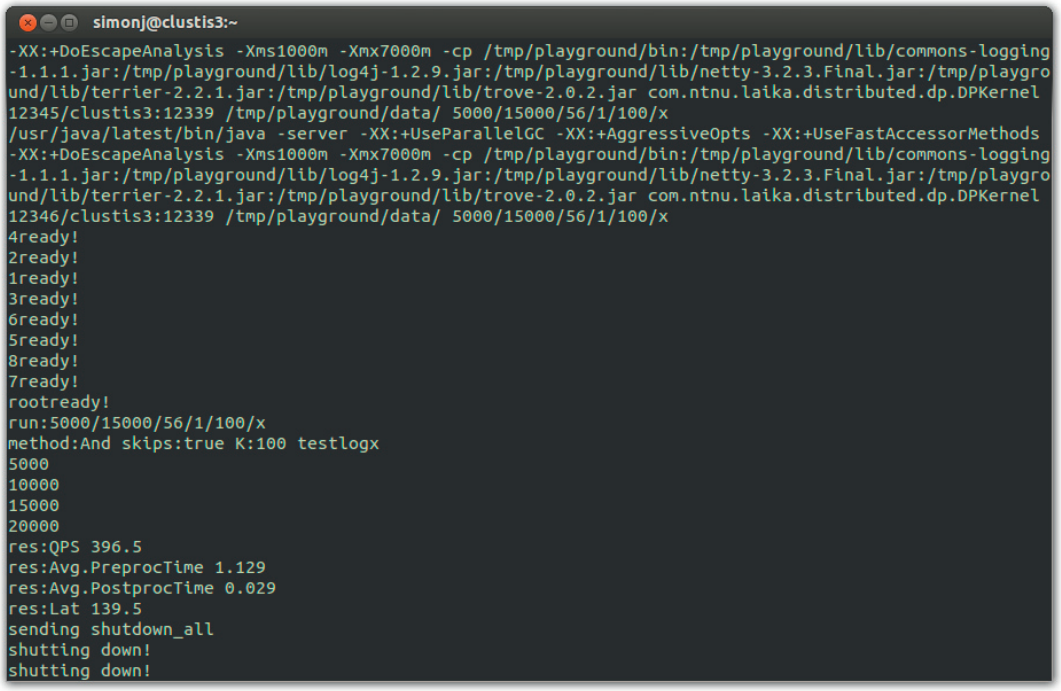



Figure 3.4: Laika in action.

saved me a lot of time. Further, this time, I used the Netty[9] API as a base for the communication manager. According to my later experience, Netty is a highly efficient and scalable communication library, which provides a lot of flexibility and very powerful features.

A slightly cleaned version of Laika corresponding to Paper A.III is available online[10]. As a recently integrated feature, it also implements query processing over a document-wise partitioned index, a screen-shot of which is shown in Figure 3.4.

Finally, Papers A.IV, A.VI and B.III led to an extension of the framework in several different directions. In particular, for Paper A.IV, I extended pipelined query processing with intra-query concurrency. For Paper A.VI, I added load-balancing for replicated terms. For Paper B.III, I integrated the LP solver originally written by Enver Kayaaslan as a part of his work on query forwarding [47].

### Simulators

Papers C.I and C.II led me to writing two simulator frameworks. For Paper C.I, I wrote a small static cache simulator and a number of tools. This work also included document processing (parsing, character removal, case folding, tokenizing, etc.) and frequency extraction from a relatively large Web crawl. For Paper C.II, we reused an event-driven simulator written by Ahmet Ridvan Duran, but eventually I rewrote most of this code.

[9] http://www.jboss.org/netty
[10] https://github.com/s-j/laika

While the query processing frameworks described above provide the most realistic results, simulation allows isolation of specific features, such as cache hits and misses, and to scale-up the system. For example, in Paper C.II, we simulate a system processing more than a million queries a day over an index containing several billions of documents. Because it does not involve actual processing, each run (equivalent to several days of the query log) takes only a few minutes of actual time. Additionally, the simulators used in Papers C.I and C.II are able to run experiments concurrently. This gave us the possibility to evaluate our ideas thoroughly and in a very short time.

### 3.5.2 Data

This section briefly summarized the data we had a chance to work with.

#### Public data

In Papers A.I–A.IV, A.VI and B.I, we use the widely known TREC GOV2 document corpus, which contains 25.2 million documents obtained in early 2004 by a crawl of `.gov` sites. These documents are truncated at 256Kb and include HTML pages, text files and the text extracted from PDF, PostScript and Word documents. In our case, the collection was indexed using Terrier and the indexed terms were cleaned from stop-words and stemmed.

In order to evaluate the efficiency, Papers A.I, A.II, A.IV and B.I use subsets of the TREC Terabyte Track 2005 Efficiency Topics, Paper A.VI uses a subset of the TREC Terabyte Track 2006 Efficiency Topics and Paper A.III uses subsets of both. The difference between the query logs is explained in Paper A.III (2005 topics correspond to Web-specific queries, while 2006 topics are more government-specific queries). The queries were preprocessed with case folding, stop-word removal and stemming and the subsets used in different papers have different number of queries. This was done in order to reduce the total execution time, i.e., the experiments that required more design-space exploration were done with shorter query logs.

Papers A.II, A.III and B.I evaluate also the quality of results. For this purpose we used the TREC Terabyte Track Adhoc Retrieval Topics and Relevance Judgments and calculated precision, recall and mean average precision. The quality of these experiments has varied. In the most recent paper, Paper A.III, we use the adhoc topics from 2004, 2005 and 2006 and perform a statistical significance test.

#### Commercial data

In papers B.III, C.I and C.II, we use Web query logs and crawls provided by Yahoo. In particular, in Paper B.III we use an index over 20 million pages sampled from the UK domain and a large query log sample. In Paper C.I we use a 300GB crawl of the UK domain from 2006 containing 37.9 million documents, and 80.7 million queries sampled from a period between April 2006 and April 2007. Finally, in Paper C.II we sample queries from five consecutive days of Yahoo! Web search and the simulated index contains several billions

of documents (document frequencies were taken from the actual index). Unfortunately, none of this data is publicly available.

#### Other

In Paper B.II, we use several datasets representing a mixture of real and synthetic data. Four of these consist of 1, 2, 3 and 4 million Twitter messages with randomly generated locations. Another data set combines texts from 20 Newsgroups and locations from LA streets. Three more data sets correspond to Wikipedia articles with spatial locations, coordinates and tags and descriptions of Flickr photos taken in the area of London, and finally, Open Street Map objects covering the entire planet.

Finally, the simulation models in Papers A.I, C.I and C.II are instantiated either with help of micro-benchmarking and data-sheets (Paper A.I) or by using the data taken from the prior work (Papers C.I and C.II).

## 3.6 Evaluation of Contributions

This section evaluates our contributions towards the main research question:

**RQ** *What are the main challenges of existing methods for efficient query processing in distributed search engines and how can these be resolved?*

Seeking to answer this question, we performed a large exploratory study described in Chapters 2 and 3 and came up with a number of contributions presented in the included papers. In the remainder of this section, we briefly summarize our contributions and show how these can be applied to a large distributed search engine.

### 3.6.1 Partitioned Query Processing

**RQ-A** *How to improve the efficiency of partitioned query processing?*

Based on our previous observations (Paper A.I), we decided to look at 2D partitioning methods, which may improve the overall performance compared to pure term-wise or document-wise partitioned indexes. Subsequently, this led us to look at pipelined query processing. We observed that this approach has several important benefits, but also a number of limitations. Therefore, our research on partitioned query processing attempted to resolve these limitations.

In Paper A.II, we came up with an extension that combines parallel disk access and pipelined query processing. This optimization is suitable for the traditional model where each posting list is read by a single disk access. Additionally, we have presented a hybrid approach that dynamically switches between semi and non-pipelined execution depending on the estimated transfer volume, and an alternative query routing heuristic. According

to our results, these optimizations significantly improve the latency of the pipelined approach.

In Paper A.III, we went in a complementary direction and came up with several skipping optimizations. In particular, we applied a skip-optimized version of the space limited pruning method by Lester et al. [94], and presented a pipelined query processing approach based on the Max-Score heuristic [150] and a further posting list assignment optimization. According to our results, these methods provide a great improvement above the baseline in terms of the performance or the quality of results.

In Paper A.IV, we improved pipelined query processing with intra-query concurrency by parallelizing both different sub-queries of the same query and different parts of the same sub-query. According to our results, these optimizations have a major impact on the query processing efficiency at lower query rates, when inter-query concurrency alone is less beneficial. At moderate query rates, we were able to reach the peak throughput of the baseline approach at nearly half of the latency.

The experiments described in these papers were carried out on a real and highly optimized distributed framework, eight query processing nodes and real data.

A question addressed, but not answered by Paper A.V was "*What happens if we resolve all of the problems, will pipelined query processing become an ultimate highly-scalable approach to distributed query processing?*". To get an insight into the answer, upon writing the thesis, we extended Laika with query processing over a document-wise partitioned index. A note summarizing these results and other findings is given in Appendix H. Unfortunately, our conclusion is that despite all our efforts and positive improvements to pipelined query processing, document-wise partitioning remains more advantageous.

### 3.6.2 Efficient Query Processing and Pruning

**RQ-B** *How to improve the efficiency of query processing and pruning?*

Our contributions in this direction relate to skipping and pruning optimizations. In Paper B.I, we presented an efficient self-skipping organization designed specifically for a document-ordered inverted index and NewPFoR compression. Further, we provided an updated version of Max-Score combining the advantage of descriptions by Turtle and Flood [150] and Strohman et al. [143], and a skipping version of the pruning method by Lester et al. [94]. According to our results with a real implementation and a relatively large document collection, both methods significantly reduce the number of processed elements and the average query latency, and narrow the efficiency gap between conjunctive (AND) and disjunctive (OR) query evaluation.

In Papers A.III and A.V, we presented a further application of skipping optimizations in pipelined query processing. As described above, these optimizations led to great improvements in the efficiency of distributed query processing.

In Paper B.III, we presented a linear programming optimization to Max-Score, which tightens score upper bounds by considering subsets of previously issued queries. Though

our current results show that the savings are not very high, we believe that this optimization can be beneficial for an index built on billions of documents. This optimization can also be used in combination with pipelined Max-Score, where it can result in even more substantial efficiency improvements.

### 3.6.3 Caching

**RQ-C** *How to improve the efficiency of caching?*

This work has been done in two directions, optimal split in static caches and result prefetching in infinite, TTL-based caches. In both papers, the results were obtained by simulation on large query logs from Yahoo! Web search.

In Paper C.I, we presented an analytical model for nearly optimal split between static result and posting list caches, which requires estimation of only few parameters.

In Paper C.II, we investigated the impact of result prefetching on the efficiency and effectiveness of large-scale Web search engines. We proposed several strategies for selecting, ordering and scheduling prefetch queries. Our offline methods rely on the queries issued on the previous day, while our online methods rely on a machine learned model. According to the results, our strategies improve the hit rate, query response time, result freshness and query degradation compared to the state-of-the-art baseline.

### 3.6.4 Connecting the Dots: Towards an Efficient Distributed Search Engine

In the following, we outline how our contributions can be combined with the architecture described in Sections 1.2 and 2.3.1. First, the search engine's frontend may combine an infinite cache for all results on disk and a smaller static cache for selected results in main memory. Second, the backend of each cluster may apply query processing over a 2D partitioned inverted index combining collection selection for document partitions and pipelined query processing over term partitions. Pipelined query processing in its turn may apply the optimizations described in our work and a posting list cache. Third, the analytical model for the optimal split between caches can be modified to account for these changes and other types of cache data as well. Finally, we may further apply the techniques described in Section 2.3 (e.g., replication, tiering and geographical query forwarding).

## 3.7 Conclusions and Further Work

We elaborated on efficient query processing in distributed search engines. We came up with several contributions on partitioned query processing, pruning and caching, which address some of their most important challenges. Our solutions were carefully designed and the experiments were done either using a real implementation or by simulation with the help of real query logs and index data. Most of the proposed techniques were found to

improve the state-of-the-art in the conducted empirical studies. However, the implications and applicability of these techniques in practice need further evaluation in real-life settings.

The process behind this thesis led to gaining knowledge and experience in distributed IR. Both in terms of the theoretical aspects and research methodology, but also in terms of actual search engine implementation. The frameworks used in our work were carefully designed and optimized for high performance. While we mainly focus on Web search, we believe that some of the problems and the ideas discussed in this thesis relate to other types of search (e.g., spatial and image search) or even to other fields (e.g., scientific computing and data mining). In what remains, we outline several directions for further work:

- The work on partitioned query processing may have many possible extensions. Pipelined query processing can be extended with dynamic load balancing and hybrid execution mentioned in Paper A.III and LP optimization similar to the one presented in Paper B.III. It is also possible to look at several alternatives, such as representing long posting lists as impact-ordered. However, while these ideas may improve the performance of pipelined query processing on a particular system, we doubt that they will improve the scalability of the method. Otherwise, a careful evaluation of scalability (with a large and varied number of nodes and collection size) is needed.
- The work on pruning can be extended by skipping not only by document ID, but also minimum required score/frequency, context/field ID, geographical location, etc., and optimizing it for the actual architecture (e.g., paying attention to the L2 cache efficiency). Skipping can also be improved by a cache where the most frequently accessed blocks can be stored uncompressed. Further, an approach that could combine the advantage of WAND and Max-Score would be interesting. Finally, it is possible to elaborate on the LP optimization described in Paper B.III.
- The work on cache modeling can be extended with other types of caches and more advanced models, while the work on result prefetching can be extended by providing a better prediction model, speculative prefetching and evaluating the economic impact with respect to power usage. We believe that prefetching has a great potential for future research.

## 3.8 Outlook

Our contributions tackle only a small subset of the currently known search engine challenges [16, 43]. There are still many unsolved problems and a large room for potential improvements. We believe that in future geographically distributed sites, result and index freshness, cloud technology, green and more intelligent power usage, machine learning techniques, personalization and integration with other types of data will become even more important. We also observe a paradigm shift from the users searching for the right content (i.e., search in indexed documents) to the content searching for the right users (i.e., user profiling and publish/subscribe systems). This transition may require a transfer of the existing technology to new problems and/or an introduction of completely new techniques.

# Bibliography

[1] S. Alici, I. S. Altingövde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy. Timestamp-based result cache invalidation for web search engines. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 973–982, Beijing, China, 2011. ACM.

[2] S. Alici, I. S. Altingövde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In *Proceedings of the 34th European Conference on Information Retrieval (ECIR)*, pages 401–412, Barcelona, Spain, 2012. Springer.

[3] I. S. Altingövde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy. Second chance: A hybrid approach for dynamic result caching in search engines. In *Proceedings of the 33rd European Conference on Information Retrieval (ECIR)*, pages 510–516, Dublin, Ireland, 2011. Springer.

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

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

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

[7] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, 2006.

[8] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 372–379, Seattle, WA, USA, 2006. ACM.

[9] V. N. Anh and A. Moffat. Pruning strategies for mixed-mode querying. In *Proceedings of the 15th International Conference on Information and Knowledge Management (CIKM)*, pages 190–197, Arlington, VA, USA, 2006. ACM.

[10] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[11] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Information Processing and Management*, 43(3):592–608, 2007.

[12] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proceedings of the 8th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 10–20, Laguna de San Rafael, Chile, 2001. IEEE Computer Society.

[13] C. Badue, R. Barbosa, P. Golgher, B. Ribeiro-Neto, and N. Ziviani. Basic issues on the processing of web queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 577–578, Salvador, Brazil, 2005. ACM.

[14] C. Badue, R. Barbosa, P. Golgher, B. Ribeiro-Neto, and N. Ziviani. Distributed processing of conjunctive queries. In *Proceedings of the ACM SIGIR 2005 Workshop on Heterogeneous and Distributed Information Retrieval (HDIR)*, pages 28–35, Salvador, Brazil, 2005.

[15] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, volume 3109 of *LNCS*, pages 400–408. Springer, 2004.

[16] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, pages 6–20, Istanbul, Turkey, 2007. IEEE Computer Society.

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

[18] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Transactions on the Web*, 2(4):20:1–20:28, 2008.

[19] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Plachouras, and L. Telloli. On the feasibility of multi-site web search engines. In *Proceedings of the 18th International Conference on Information and Knowledge Management (CIKM)*, pages 425–434, Hong Kong, China, 2009. ACM.

[20] R. Baeza-Yates and S. Jonassen. Modeling static caching in web search engines. In *Proceedings of the 34th European Conference on Information Retrieval (ECIR)*, pages 436–446, Barcelona, Spain, 2012. Springer.

[21] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 74–85, Santiago, Chile, 2007. Springer.

[22] R. Baeza-Yates, C. Middleton, and C. Castillo. The geographical life of search. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), Vol. 1*, pages 252–259, Milano, Italy, 2009. IEEE Computer Society.

[23] R. Baeza-Yates, V. Murdock, and C. Hauff. Efficiency trade-offs in two-tier web search systems. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 163–170, Boston, MA, USA, 2009. ACM.

[24] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second ed.* Addison-Wesley Professional, 2011.

[25] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):4:1–4:18, 2008.

[26] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Experimental Algorithms*, volume 4007 of *LNCS*, pages 146–157. Springer, 2006.

[27] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[28] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 321–328, Sheffield, UK, 2004. ACM.

[29] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 82–89, Geneva, Switzerland, 2010. ACM.

[30] R. Blanco, B. B. Cambazoglu, F. Junqueira, I. Kelly, and V. Leroy. Assigning documents to master sites in distributed search. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 67–76, Glasgow, Scotland, UK, 2011. ACM.

[31] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 25–28, Buenos Aires, Argentina, 2005. Springer.

[32] P. Boldi and S. Vigna. MG4J at TREC 2005. In *Proceedings of the 14th Text REtrieval Conference (TREC)*, Gaithersburg, MD, USA, 2005. National Institute of Standards and Technology.

[33] C. Bonacic, C. Garcia, M. Marin, M. Prieto, and F. Tirado. Exploiting hybrid parallelism in web search engines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, pages 414–423, Las Palmas de Gran Canaria, Spain, 2008. Springer.

[34] C. Bonacic, C. Garcia, M. Marin, M. Prieto, F. Tirado, and C. Vicente. Improving search engines performance on multithreading processors. In *Proceedings of the 8th International Conference on High Performance Computing for Computational Science (VECPAR)*, pages 201–213, Toulouse, France, 2008. Springer.

[35] E. Bortnikov, R. Lempel, and K. Vornovitsky. Caching for realtime search. In *Proceedings of the 33rd European Conference on Information Retrieval (ECIR)*, pages 104–116, Dublin, Ireland, 2011. Springer.

[36] U. Brefeld, B. B. Cambazoglu, and F. Junqueira. Document assignment in multi-site search engines. In *Proceedings of the 4th International Conference on Web Search and Data Mining (WSDM)*, pages 575–584, Hong Kong, China, 2011. ACM.

[37] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM)*, pages 426–434, New Orleans, LA, USA, 2003. ACM.

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

[39] S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In *Proceedings of the 16th International Conference on Information and Knowledge Management (CIKM)*, pages 761–770, Lisbon, Portugal, 2007. ACM.

[40] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[41] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, WA, USA, 1995. ACM.

[42] B. B. Cambazoglu and C. Aykanat. A term-based inverted index organization for communication-efficient parallel query processing. In *IFIP International Conference on Network and Parallel Computing*, Tokyo, Japan, 2006.

[43] B. B. Cambazoglu and R. Baeza-Yates. Scalability challenges in web search engines. In *Advanced Topics in Information Retrieval*, volume 33, pages 27–50. Springer, 2011.

[44] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *Proceedings of the 21st International Conference on Computer and Information Sciences (ISCIS)*, pages 717–725, Istanbul, Turkey, 2006. Springer.

[45] B. B. Cambazoglu, F. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, pages 181–190, Raleigh, NC, USA, 2010. ACM.

[46] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 411–418, Boston, MA, USA, 2009. ACM.

[47] B. B. Cambazoglu, E. Varol, E. Kayaaslan, C. Aykanat, and R. Baeza-Yates. Query forwarding in geographically distributed search engines. In *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 90–97, Geneva, Switzerland, 2010. ACM.

[48] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the 3rd International Conference on Web Search and Data Mining (WSDM)*, pages 411–420, New York, NY, USA, 2010. ACM.

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

[50] Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 277–288, Chicago, IL, USA, 2006. ACM.

[51] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *Proceedings of the 1st International Conference on Web Search and Data Mining (WSDM)*, pages 15–24, Palo Alto, CA, USA, 2008. ACM.

[52] A. Chowdhury and G. Pass. Operational requirements for scalable search systems. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM)*, pages 435–442, New Orleans, LA, USA, 2003. ACM.

[53] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 423–432, Glasgow, Scotland, UK, 2011. ACM.

[54] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[55] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient IR using array databases. *The VLDB Journal*, 17(1):151–168, 2008.

[56] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29(1):1:1–1:25, 2010.

[57] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 656–665, Cancun, Mexico, 2008. IEEE Computer Society.

[58] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[59] R. Delbru, S. Cmapinas, K. Samp, and G. Tummarello. Adaptive frame of reference for compressing inverted lists. Technical report, DERI Galway, 2010.

[60] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Revised Papers from the 3rd International Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 91–104, Washington, DC, USA, 2001. Springer.

[61] B. Ding and A. C. König. Fast set intersection in memory. *Proceedings of the VLDB Endowment*, 4(4):255–266, 2011.

[62] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proceedings of the 4th International Conference on Web Search and Data Mining (WSDM)*, pages 137–146, Hong Kong, China, 2011. ACM.

[63] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high-performance IR query processing. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 1213–1214, Beijing, China, 2008. ACM.

[64] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, pages 421–430, Madrid, Spain, 2009. ACM.

[65] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 993–1002, Beijing, China, 2011. ACM.

[66] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 102–113, Santa Barbara, CA, USA, 2001. ACM.

[67] T. Fagni, R. Perego, and F. Silvestri. A highly scalable parallel caching system for web search engine results. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing*, pages 347–354, Pisa, Italy, 2004. Springer.

[68] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, 24(1):51–78, 2006.

[69] E. Feuerstein, V. Gil-Costa, M. Mizrahi, and M. Marin. Performance evaluation of improved web search algorithms. In *Revised Selected Papers from the 9th International Conference on High Performance Computing for Computational Science (VECPAR)*, pages 236–250, Berkeley, CA, USA, 2011. Springer.

[70] E. Feuerstein, M. Marin, M. Mizrahi, V. Gil-Costa, and R. Baeza-Yates. Two-dimensional distributed inverted files. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 206–213, Saariselkä, Finland, 2009. Springer.

[71] M. Fontoura, M. Gurevich, V. Josifovski, and S. Vassilvitskii. Efficiently encoding term co-occurrences in inverted indexes. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 307–316, Glasgow, Scotland, UK, 2011. ACM.

[72] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[73] A. B. Fossaa, S. Jonassen, J. M. W. Kristiansen, and O. Natvig. Solbrille : Bringing back the TIME. Technical report, Norwegian University of Science and Technology, 2009.

[74] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 12(4):441–460, 2009.

[75] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, pages 431–440, Madrid, Spain, 2009. ACM.

[76] A. Greenberg and S. Branson. Clustering web search results using suffix tree methods. Technical report, Stanford University, 2002.

[77] S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC Terabyte Track. In *Proceedings of the 15th Text REtrieval Conference (TREC)*, pages 1–12, Gaithersburg, MD, USA, 2006. National Institute of Standards and Technology.

[78] U. Hölzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[79] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[80] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To share or not to share? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 351–362, Vienna, Austria, 2007. VLDB Endowment.

[81] S. Jonassen. Distributed inverted indexes. Master's thesis, Norwegian University of Science and Technology, 2008.

[82] S. Jonassen. Scalable search platform: improving pipelined query processing for distributed full-text retrieval. In *Proceedings of the 21st International World Wide Web Conference, Companion Volume (WWW Comp.)*, pages 145–150, Lyon, France, 2012. ACM.

[83] S. Jonassen and S. E. Bratsberg. Impact of the query model and system settings on performance of distributed inverted indexes. In *Proceedings of the 22nd Norwegian Informatics Conference (Norsk Informatikkonferanse, NIK)*, pages 143–154, Trondheim, Norway, 2009. Tapir Akademisk Forlag.

[84] S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *Proceedings of the 11th International Conference on Web Information Systems Engineering (WISE)*, pages 587–601, Hong Kong, China, 2010. Springer.

[85] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Proceedings of the 33rd European Conference on Information Retrieval (ECIR)*, pages 530–542, Dublin, Ireland, 2011. Springer.

[86] S. Jonassen and S. E. Bratsberg. Improving the performance of pipelined query processing with skipping. In *Proceedings of the 13th International Conference on Web Information Systems Engineering (WISE)*, pages 1–15, Paphos, Cyprus, 2012. Springer.

[87] S. Jonassen and S. E. Bratsberg. Intra-query concurrent pipelined processing for distributed full-text retrieval. In *Proceedings of the 34th European Conference on Information Retrieval (ECIR)*, pages 413–425, Barcelona, Spain, 2012. Springer.

[88] S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 631–640, Portland, OR, USA, 2012. ACM.

[89] E. Kayaaslan, B. B. Cambazoglu, R. Blanco, F. Junqueira, and C. Aykanat. Energy-price-driven query processing in multi-center web search engines. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 983–992, Beijing, China, 2011. ACM.

[90] P. Lacour, C. Macdonald, and I. Ounis. Efficiency comparison of document matching techniques. In *Proceedings of the Efficiency Issues in Information Retrieval Workshop at the 30th European Conference on Information Retrieval (ECIR)*, pages 37–46, Glasgow, Scotland, UK, 2008. Springer.

[91] H. T. Lam, R. Perego, N. T. Quan, and F. Silvestri. Entry pairing in inverted file. In *Proceedings of the 10th International Conference on Web Information Systems Engineering (WISE)*, pages 511–522, Poznan, Poland, 2009. Springer.

[92] A. N. Langville and C. D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

[93] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, pages 19–28, Budapest, Hungary, 2003. ACM.

[94] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of the 6th International Conference*

*on Web Information Systems Engineering (WISE)*, pages 470–477, New York, NY, USA, 2005. Springer.

[95] H. Li, W.-C. Lee, A. Sivasubramaniam, and C. L. Giles. A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proceedings of the 7th International Conference on Web Engineering (ICWE)*, pages 121–136, Como, Italy, 2007. Springer.

[96] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer, 2006.

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

[98] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, pages 257–266, Chiba, Japan, 2005. ACM.

[99] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Proceedings of the 2nd International Conference on Scalable Information Systems (InfoScale)*, pages 43:1–43:9, Suzhou, China, 2007. ICST.

[100] Y.-C. Ma, C.-P. Chung, and T.-F. Chen. Load and storage balanced posting file partitioning for parallel information retrieval. *Journal of Systems and Software*, 84(5):864–884, 2011.

[101] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 209–220, A Coruña, Spain, 2000. IEEE Computer Society.

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

[103] M. Marin, F. Ferrarotti, M. Mendoza, C. Gomez-Pantoja, and V. Gil-Costa. Location cache for web queries. In *Proceedings of the 18th International Conference on Information and Knowledge Management (CIKM)*, pages 1995–1998, Hong Kong, China, 2009. ACM.

[104] M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *Proceedings of the 16th International Conference on Information and Knowledge Management (CIKM)*, pages 935–938, Lisbon, Portugal, 2007. ACM.

[105] M. Marin, V. Gil-Costa, C. Bonacic, R. Baeza-Yates, and I. Scherson. Sync/async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 36(4):153–168, 2010.

[106] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th International Symposium on High Performance Distributed Computing (HPDC)*, pages 215–226, Chicago, IL, USA, 2010. ACM.

[107] M. Marin and C. Gomez. Load balancing distributed inverted files. In *Proceedings of the 9th Annual International Workshop on Web Information and Data Management (WIDM)*, pages 57–64, Lisbon, Portugal, 2007. ACM.

[108] M. Marin, C. Gomez-Pantoja, S. Gonzalez, and V. Gil-Costa. Scheduling intersection queries in term partitioned inverted files. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, pages 434–443, Las Palmas de Gran Canaria, Spain, 2008. Springer.

[109] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[110] R. M. C. McCreadie, C. Macdonald, and I. Ounis. Comparing distributed indexing: To MapReduce or not? In *Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 41–48, Boston, MA, USA, 2009.

[111] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 348–355, Seattle, WA, USA, 2006. ACM.

[112] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007.

[113] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proceedings of the 10th International Conference on Data Engineering (ICDE)*, pages 428–437, Houston, TX, USA, 1994. IEEE Computer Society.

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

[115] A. Moffat and J. Zobel. What does it mean to "measure performance"? In *Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE)*, pages 1–12, Brisbane, Australia, 2004. Springer.

[116] O. Natvig. Compression in XML search engines. Master's thesis, Norwegian University of Science and Technology, 2010.

[117] G. Navarro and S. J. Puglisi. Dual-sorted inverted lists. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 309–321, Los Cabos, Mexico, 2010. Springer.

[118] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and D. Johnson. Terrier information retrieval platform. In *Proceedings of the 27th European Conference on Information Retrieval (ECIR)*, pages 517–519, Santiago de Compostela, Spain, 2005. Springer.

[119] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engines. *Information Processing and Management*, 48(5):828–840, 2011.

[120] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static query result caching revisited. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 1169–1170, Beijing, China, 2008. ACM.

[121] R. Ozcan, I. S. Altingövde, and Ö Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):9:1–9:25, 2011.

[122] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale)*, pages 1:1–1:7, Hong Kong, China, 2006. ACM.

[123] M. Persin, J. Zobel, and R. Sacks-Davis. Fast document ranking for large scale information retrieval. In *Proceedings of the 1st International Conference on Applications of Databases (ADB)*, pages 253–266, Vadstena, Sweden, 1994. Springer.

[124] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[125] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[126] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale)*, pages 34:1–34:8, Hong Kong, China, 2006. ACM.

[127] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd Conference on Digital Libraries (DL)*, pages 182–190, Pittsburgh, PA, USA, 1998. ACM.

[128] K. M. Risvik. *Scaling Internet Search Engines - Methods and Analysis*. PhD thesis, Norwegian University of Science and Technology, 2004.

[129] K. M. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *Proceedings of the 1st Conference on Latin American Web Congress (LA-WEB)*, pages 132–, Sanitago, Chile, 2003. IEEE Computer Society.

[130] K. M. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, 39(3):289–302, 2002.

[131] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, M. Gatford, and A. Payne. Okapi at TREC-4. In *Proceedings of the 4th Text REtrieval Conference (TREC)*, pages 73–96, Gaithersburg, MD, USA, 1996. National Institute of Standards and Technology.

[132] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *Proceedings of the 12th International Conference on Advances in Spatial and Temporal Databases (SSTD)*, pages 205–222, Minneapolis, MN, USA, 2011. Springer.

[133] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 51–58, New Orleans, LA, USA, 2001. ACM.

[134] D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *Proceedings of the 5th International Conference on Web Search and Data Mining (WSDM)*, pages 423–432, Seattle, WA, USA, 2012. ACM.

[135] F. Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on Information Retrieval (ECIR)*, pages 101–112, Rome, Italy, 2007. Springer.

[136] F. Silvestri, T. Fagni, S. Orlando, P. Palmerini, and R. Perego. A hybrid strategy for caching web search engine results. In *Proceedings of The 12th International World Wide Web Conference (WWW Posters)*, Budapest, Hungary, 2003.

[137] F. Silvestri and R. Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th International Conference on Information and Knowledge Management (CIKM)*, pages 1219–1228, Toronto, ON, Canada, 2010. ACM.

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

[139] G. Skobeltsyn, T. Luu, I. Podnar Žarko, M. Rajman, and K. Aberer. Query-driven indexing for scalable peer-to-peer text retrieval. *Future Generation Computer Systems*, 25(1):89–99, 2009.

[140] O. Sornil and E. A. Fox. Hybrid partitioned inverted indices for large-scale digital libraries. In *Proceedings of the 4th International Conference on Asian Digital Libraries (ICADL)*, Bangalore, India, 2001.

[141] A. Spink, D. Wolfram, M. B. J. Jansen, and T. Saracevic. Searching the web: the public and their queries. *Journal of the American Society for Information Science and Technology*, 52(3):226–234, 2001.

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

[143] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 219–225, Salvador, Brazil, 2005. ACM.

[144] S. Tatikonda, B. B. Cambazoglu, and F. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 963–972, Beijing, China, 2011. ACM.

[145] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 738–739, Boston, MA, USA, 2009. ACM.

[146] J. Teevan, E. Adar, R. Jones, and M. A. S. Potts. Information re-retrieval: repeat queries in Yahoo's logs. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 151–158, Amsterdam, The Netherlands, 2007. ACM.

[147] A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in shared nothing text document information retrieval systems. *The VLDB Journal*, 2(3):243–275, 1993.

[148] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *Proceedings of the 16th International Conference on Information and Knowledge Management (CIKM)*, pages 987–990, Lisbon, Portugal, 2007. ACM.

[149] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proceedings of the VLDB Endowment*, 2(1):838–849, 2009.

[150] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[151] W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, University of Melbourne, 2007.

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

[153] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[154] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images, Second ed.* John Wiley & Sons, Inc., 1999.

[155] W. Xi, O. Sornil, M. Luo, and E. A. Fox. Hybrid partition inverted files: Experimental validation. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, pages 422–431, Roma, Italy, 2002. Springer.

[156] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 147–154, Boston, MA, USA, 2009. ACM.

[157] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, pages 401–410, Madrid, Spain, 2009. ACM.

[158] X. Yinglian and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of the 21 Annual Joint Conference of the IEEE*

*Computer and Communications Societies (INFOCOM)*, volume 3, pages 1238–1247. IEEE, 2002.

[159] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 688–699, Shanghai, China, 2009. IEEE Computer Society.

[160] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 521–532, Long Beach, CA, USA, 2010. IEEE Computer Society.

[161] F. Zhang, S. Shi, H. Yan, and J.-R. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *Proceedings of the 3rd International Conference on Web Search and Data Mining (WSDM)*, pages 371–380, New York, NY, USA, 2010. ACM.

[162] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 387–396, Beijing, China, 2008. ACM.

[163] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Long Beach, CA, USA, 2007. IEEE.

[164] M. Zhu, S. Shi, M. Li, and J.-R. Wen. Effective top-k computation with term-proximity support. *Information Processing and Management*, 45(4):401–412, 2009.

[165] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley (Reading MA), 1949.

[166] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6:1–6:56, 2006.

[167] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 59–, Atlanta, GA, USA, 2006. IEEE Computer Society.

# Included Papers

*"If we knew what we were doing,*
*it wouldn't be called research."*

– Albert Einstein

This part contains the papers that constitute the main body of the thesis. The format of the text, tables and figures has been altered in order to improve readability.

Appendix A

# Paper A.II: A Combined Semi-Pipelined Query Processing Architecture for Distributed Full-Text Retrieval

Simon Jonassen and Svein Erik Bratsberg.
*Appeared at the 11th International Conference on Web Information Systems Engineering (WISE), Hong Kong, China, December 2010.*

**Abstract:** Term-partitioning is an efficient way to distribute a large inverted index. Two fundamentally different query processing approaches are pipelined and non-pipelined. While the pipelined approach provides higher query throughput, the non-pipelined approach provides shorter query latency. In this work we propose a third alternative, combining non-pipelined inverted index access, heuristic decision between pipelined and non-pipelined query execution and an improved query routing strategy. From our results, the method combines the advantages of both approaches and provides high throughput and short query latency. Our method increases the throughput by up to 26% compared to the non-pipelined approach and reduces the latency by up to 32% compared to the pipelined.

## A.1 Introduction

Index organization and query processing are two of the most central and challenging areas within distributed information retrieval. Two fundamental distributed index organization methods for full-text indexing and retrieval are term- and document-based partitioning. With document-based partitioning each node stores a local inverted index for its own subset of documents. With term-partitioning each node stores a subset of a global inverted index. A large number of studies [1, 2, 7, 11, 12, 13, 15] have tried to compare these two organizations to each other. Each method has its advantages and disadvantages, and according to the published results, both methods may provide superior performance under different circumstances.

The main problems of document-based partitioning include a large number of query messages and inverted list accesses (as disk-seeks and lexicon look-ups) for each query. Term-partitioning on the other hand reduces the number of query messages, improves inverted index accesses and provides more potential for concurrent queries, but it faces two difficult challenges: a high network load and a risk for a bottleneck at the *query processing node*.

Both challenges arise from the fact that the posting lists corresponding to the terms occurring in the same query may be assigned to two or more different nodes. In this case a simple solution is to transfer each list to the query processing node and then process the posting lists. We refer to this solution as *a non-pipelined approach*. However, for a large document collection a single compressed posting list can occupy hundreds of megabytes and this processing model incurs high network load. Further, as a single node is chosen to process the posting lists the performance of this node is critical. Both challenges become more prominent as the document collection grows.

In order to solve these challenges, Moffat et al. [13] have proposed a pipelined architecture for distributed text-query processing. The main idea behind the method is to process a query gradually, by one node at a time. The performance improvement is achieved from work distribution and a dynamic space-limited pruning method [5].

Although the pipelined architecture reduces the network and processing load, it does not necessarily reduce the average query latency. In our opinion, considering query latency is important under evaluation of the system throughput, and with the pipelined approach the average query latency will be significantly longer. However, neither of the two papers [12, 13] describing the pipelined approach look at the query latency, but only at the normalized query throughput and system load.

The contribution of this work is as follows. We look at query latency aspects of the pipelined and non-pipelined query processing approaches. We propose a novel query processing approach, which combines non-pipelined disk accesses, a heuristic method to choose between pipelined and non-pipelined posting-list processing, and an improved query routing strategy. We evaluate the improvement achieved with the proposed method with the GOV2 document collection and a large TREC query set. We show that our method outperforms both the pipelined and the nasdon-pipelined approaches in the measured throughput and latency.

This paper is organized as follows. Section A.2 gives a short overview of related work and limits the scope of this paper. Section A.3 describes the pruning method and state of the art distributed query processing approaches. Section A.4 presents our method. The experimental framework and results are given in Section A.5. The final conclusions and directions for the further work follow in Section A.6.

## A.2 Related Work

A performance-oriented comparison between term- and document-partitioned distributed inverted indexes have been previously done by a large number of publications [1, 2, 7, 11, 12, 13, 15]. Most of these refer to term-based partitioning as a method with good potential, but practically complicated due to high network load and CPU/disk load imbalance.

The non-pipelined approach is an ad-hoc method considered by most of the books and publications within distributed IR. The pipelined approach was first presented by Moffat et al. [13]. The method is based on an efficient pruning algorithm for space-limited query evaluation originally presented by Lester et al. [5] and succeeds to provide a higher query throughput than the non-pipelined approach. However, due to load imbalance the method alone is unable to outperform a corresponding query processing method for a document-partitioned index. The load balancing issues of the pipelined approach have been further considered by Moffat et al. [12] and Lucchese et al. [6].

While Moffat et al. looked only at maximizing query throughput, Lucchese et al. have combined both throughput and latency in a single optimization metric. In our work we minimize query latency and maximize query throughput from the algorithmic perspective alone, with no consideration to posting list assignment.

As an alternative to these methods, a large number of recent publications from Yahoo [8, 9, 10, 11] study processing of conjunctive queries with a term-partitioned inverted index. However, from the description provided by Marin et al. [9] the methods considered in their work are based on impact-ordered inverted lists and bulk-synchronous programming model. Therefore, we do not consider this type of queries nor posting ordering in our work.

## A.3 Preliminaries

In this paper we look at processing of disjunctive queries with a distributed term-partitioned disk-stored document-ordered inverted index [17]. Each posting list entry represents a document ID and *term frequency* $f_{d,t}$ denoting the number of occurrences. Consecutive document IDs are gap-coded and compressed with Gamma coding, term frequencies are encoded with unary coding. The vocabulary is also stored on disk. For each term it stores a posting list pointer and the corresponding *collection frequency* $F_t$ and *document frequency* $f_t$. Document lengths and other global index statistics are stored in main memory.

Query processing is generally done by stemming and stop-word processing the query, looking-up the vocabulary, fetching and processing the corresponding posting list, followed by a final extraction, sorting and post-processing of the $K$ best results.

### Space-Limited Adaptive Pruning

In order to speed-up query processing an efficient pruning technique has been presented by Lester et al. [5]. The main idea is to process posting lists *term-at-a-time* according to an increasing $F_t$. For each posting the algorithm may either create a new or update an existing *accumulator* (a partial document score). Some of the existing accumulators might also be removed if their partial scores are too low. Further, the algorithm uses two threshold variables, *frequency threshold value* $h_t$ and *score threshold value* $v_t$ in order to restrict the maximum number of maintained accumulators by a *target value* $L$.

With a distributed term-partitioned inverted index each node stores a number of complete posting lists. All queries submitted to the system are forwarded to one of the nodes chosen in a round robin manner, which performs stemming and stop-word removal. To speed-up processing, each node stores a complete vocabulary. The set of stems corresponding to the query is further partitioned into a number of *sub-queries*, where each sub-query contains terms assigned to the same node.

### Non-Pipelined Approach

Each sub-query can then be transferred to the node maintaining the corresponding posting lists. Each of these nodes will fetch compressed posting lists and send these back to the node responsible for the processing of the query, *the query processing node*. The query processing node will further receive all of the posting lists, decompress and process these according to the method described earlier. In order to reduce the network load the query processing node is chosen to be the node maintaining the longest inverted list.

### Pipelined Approach

The non-pipelined method has two main problems: high network load and heavy workload on the query processing node. Alternatively to this approach, query processing can be done by routing a *query bundle* through all of the nodes storing the query terms. The query bundle is created by the node receiving the query and includes the original query, a number of sub-queries, routing information and an accumulator data structure.

Because of the pruning method the route is chosen according to increasing minimum $F_t$ in each sub-query. When a node receives the query bundle it fetches and decompresses the corresponding posting lists and combines these with the existing accumulators according to the pruning method described earlier. Further, it stores the updated accumulator set into the bundle and forwards it to the next node. The last node in the route has to extract and post-process the top-results and return these to the query broker.

In addition to reduced processing cost due to pruned postings and a limited number of accumulators, the combination of the pruning method with the pipelined distributed query processing gives the following advantages:

1. Processing of a single query is distributed across a number of nodes. Any node receiving a bundle has to combine its posting lists with an existing result set, which is limited to $L$. The last node in the route has additionally to perform extraction of top-results, where the number of candidates is also limited to $L$.
2. The upper bound for the size of a query bundle transferred between any two nodes is proportional to $L$, which can be used to reduce the network load.

## A.4 A Combined Semi-Pipelined Approach

The pipelined approach has clearly a number of advantages compared to the non-pipelined approach. These include a more even distribution of work over all of the nodes involved in a query and reduced network and CPU loads. However, the method has a number of issues we want to address in this paper:

1. Non-parallel disk-accesses and overheads associated with query bundle processing result in long query latencies.
2. An accumulator set has less compression potential than an inverted list.
3. The pipelined approach does not always provide benefits compared to the non-pipelined.
4. The routing strategy does not minimize the number of transferred accumulators.

Now we are going to explain these problems in more detail and present three methods to deal with one or several of these at a time. The first method reduces the average query latency by performing fetching and decompression of posting lists in parallel. For the second method that deals with second and third issue, we use a decision heuristic to choose whether a query should be executed as a semi-pipelined or a non-pipelined query. As the third method, we apply an alternative routing strategy to the original $\min(F_t)$ to deal with the last issue. In the next section we show that a combination of all three methods provides the advantages of both the pipelined and non-pipelined approaches and significantly improves the resulting system performance.

### A.4.1 Semi-Pipelined Query Processing

With our first technique we are trying to combine latency hiding advantages of the non-pipelined approach with load reduction and distribution of the pipelined method. Using an idea similar to Lucchese et al. [6], we can express the query latency associated with the non-pipelined approach as:

$$T_{\mathrm{non-pl}}(q) = T_{\substack{\mathrm{query} \\ \mathrm{overhead}}} + \max_{q_i \in q} \sum_{t \in q_i} (T_{\mathrm{disk}}(f_t\sigma) + T_{\mathrm{trans}}(f_t\sigma)) + T_{\mathrm{process}}(q) \tag{A.1}$$

$$T_{\text{process}}(q) = \sum_{t \in q} (T_{\text{decomp}}(f_t) + T_{\text{compute}}(f_t, L)) + T_{\text{top}}(K, L) \tag{A.2}$$

Where $q$ is a set of sub-queries $\{q_1, ., q_l\}$. $T_{\text{disk}}$ is the time to fetch an inverted list of $f_t$ postings with an average compressed size of $\sigma$ bytes per posting. $T_{\text{decomp}}$ is the time to decompress $f_t$ postings. $T_{\text{compute}}$ is the time to merge (join) $f_t$ postings into an accumulator set of $L$ postings, and $T_{\text{top}}$ is the time to extract, sort and post-process the $K$ best results out of $L$ candidates. $T_{\text{trans}}$ is the time to transfer a data structure, in this case a posting list of size $f_t\sigma$. We include the transfer time into Eq. A.1, since due to the Zipf's Law [16] it is reasonable to assume that transfer times of shorter sub-queries can be hidden within disk access times of the longest one. In the case when the assumption does not hold or with expensive network receive calls, the transfer time must be included into Eq. A.2 instead.

Finally, we use $T_{\underset{\text{overhead}}{\text{query}}}$ to denote the time spent on transfer of the query and sub-queries itself, lexicon look-up, and result output. In future discussion we also neglect the time spent on transfer of an additional network message per sub-query, as this time is insignificantly small compared to the time spent on posting list fetching, transfer and processing.

Further, the query latency associated with the pipelined approach can be expressed as:

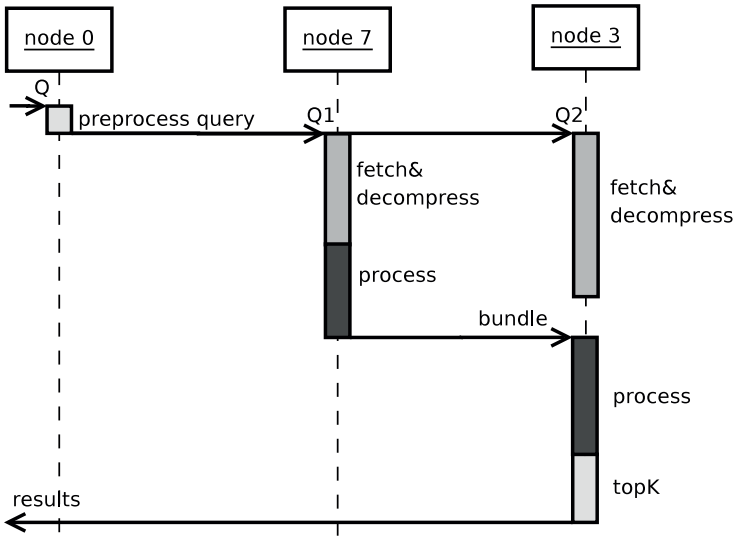
$$T_{\text{pl}}(q) = T_{\underset{\text{overhead}}{\text{query}}} + \sum_{q_i \in q} (\sum_{t \in q_i} T_{\text{disk}}(f_t\sigma) + T_{\underset{\text{overhead}}{\text{bundle}}^{i,l}}(L)) + T_{\text{process}}(q) \tag{A.3}$$

Where $T_{\text{process}}$ is defined by Eq. A.2 and the bundle overhead can be expressed as the sum of compression, decompression and transfer of the accumulator set with a compression size of $\tau$ bytes per element:

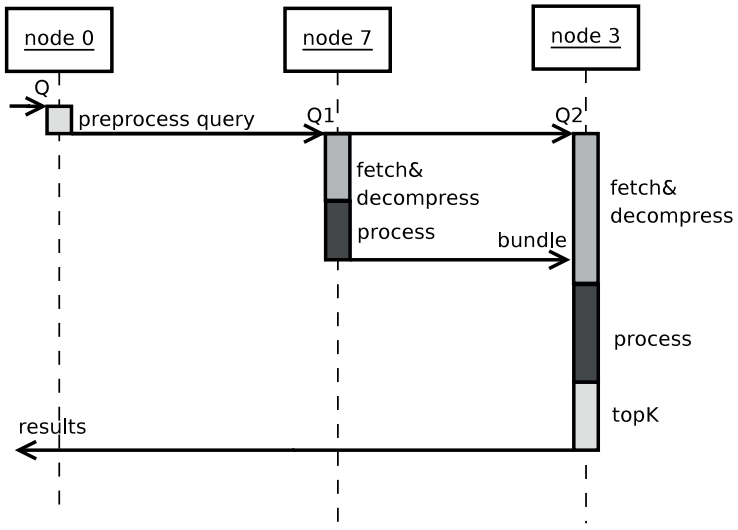
$$T_{\underset{\text{overhead}}{\text{bundle}}^{i,l}}(L) = \begin{cases} T_{\text{comp}}(L) + T_{\text{trans}}(L\tau) & \text{if } i = 1 \\ T_{\text{comp}}(L) + T_{\text{trans}}(L\tau) + T_{\text{decomp}}(L) & \text{if } i \neq 1, l \\ T_{\text{decomp}}(L) & \text{if } i = l \end{cases} \tag{A.4}$$

Our idea is to use a different execution approach, where all of the posting-list fetches and posting list decompressions can be done in parallel for different sub-queries. Further, we can start a pipelined result accumulation with a query bundle as soon as all the data required for the first (shortest) sub-query is fetched and decompressed.

For a query consisting of only two sub-queries we have two possible scenarios. In the first case, as we illustrate in Figure A.1(a), the node receiving the bundle has already fetched and decompressed the data, it can proceed straight to merging the posting data with the accumulated results. In this case, any latency associated with disk-access and decompression at the second node is hidden within posting list processing and query bundle overhead at the first node. Otherwise, as we illustrate in Figure A.1(b), the node receiving the bundle still needs to wait for the data. In this case, the query processing latency at the first node is hidden within the disk-access and decompression latency at the second.




(a) Disk-Access/Decompression Latency Hiding




(b) Processing Latency Hiding

Figure A.1: Latency hiding with the semi-pipelined approach.

According to this logic, for an arbitrary number of sub-queries the latency can be expressed as:

$$\begin{aligned}
T_{\text{semi-pl},i}(q) &= \max(\sum_{t \in q_i}(T_{\text{disk}}(f_t\sigma) + T_{\text{decomp}}(f_t)), T_{\text{semi-pl},i-1}(q)) \\
&+ \sum_{t \in q_i} T_{\text{compute}}(f_t, L) + T_{\substack{\text{bundle} \\ \text{overhead}}i,l}(L) \\
T_{\text{semi-pl},1}(q) &= T_{\substack{\text{query} \\ \text{overhead}}} + \sum_{t \in q_1}(T_{\text{disk}}(f_t\sigma) + T_{\text{decomp}}(f_t) + T_{\text{compute}}(f_t, L)) \\
&+ T_{\substack{\text{bundle} \\ \text{overhead}}1,l}(L) \\
T_{\text{semi-pl}}(q) &= T_{\text{semi-pl},l}(q) + T_{\text{top}}(K, L)
\end{aligned} \tag{A.5}$$

Eq. A.5 is hard to generalize without looking at the exact data distribution, compression ratios and processing, transfer and disk-access costs. However, from the equation we see that the method provides a significantly shorter query latency compared to the pipelined approach. The method is also comparable to the non-pipelined approach under disk-expensive settings and provides a significant improvement under network- and CPU-expensive settings. In addition to the model presented so far, which does not account for concurrent query execution, our approach provides the same load reduction and distribution as the pipelined method.

We present our approach in Alg. A.1-A.2. The node assembling the query bundle sends $l - 1$ replicas of the bundle marked as a *fake* to each of the sub-query processing nodes, except from the first node which receives the query bundle itself. The first node receiving the original query bundle fetches the posting lists and starts processing as normal. For any other node there are two different scenarios similar to the ones we have described earlier. With either of these the node will fetch and decompress the corresponding posting lists, extract the accumulator set and perform further query processing. Finally, the query bundle is either forwarded to the next node or the final results are extracted, post-processed and returned to the query broker.

**Algorithm A.1**: processQuery($q$)

```
1 preprocess q;
2 partition q into {q_1, .., q_l} and sort these by increasing min_{t∈q_i} F_t order;
3 create a new query bundle b;
4 mark b as real;
5 for sub-query q_i do
6     set b's counter idx to i;
7     if i ≠ 1 and b is marked as real then mark b as fake;
8     send b to the corresponding node;
```

**Algorithm A.2**: processBundle($b$)

```
1  if b is marked as fake then
2      for t ∈ q_idx do fetch and decompress I_t;
3      if the corresponding accumulator set A is received then
4          combine the posting data with A;
5          if idx = l then
6              extract, sort and post-process the K best results; send the results to the query
               broker;
7          else
8              put A into b; increment idx; send b to the next node in the route;
9  else if b is marked as real then
10     if idx = 1 then for t ∈ q_idx do fetch and decompress I_t;
11     extract A from the b;
12     if all the required I_t are fetched and decompressed then
13         similar to lines 4-8;
14
```

## A.4.2 Combination Heuristic

Besides the latency part, we can look at the next two observations on our list. First, document frequency values stored in postings tend to be small integers which can be compressed with unary coding or similar. The document IDs in postings are gap-coded and then compressed with Gamma coding. Thus long, dense posting lists achieve a high compression ratio. Document IDs in a pruned accumulator set tend to be sparse and irregular. Partial similarity scores stored in an accumulator data structure are single or double precision floats which cannot be compressed as good as small integer values. Therefore, even if it is possible to reduce the number of postings/accumulators transferred between two nodes, the data volume might not be reduced. Additional costs associated with compression/decompression must also be kept in mind.

Second, the pipelined or semi-pipelined approach itself will not necessarily reduce the number of scored postings or maintained accumulators. If none of the posting lists are longer than $L$, the algorithm will not give any savings for network or CPU loads. The network load will actually increase if the same data is transferred between every pair of nodes in the query route.

From these observations we suggest that, in order to reduce the query latency and network load, each query might be executed in either a non-pipelined or a semi-pipelined way. To choose between the two approaches we look at the amount of data transferred between the nodes. The upper bound for the amount of data can be approximated by Eq. A.6.

$$\hat{V}_{\mathrm{semi-pl}} = \tau \sum_{j=1}^{j=l-1} \min(L, \sum_{\substack{t\in \\ \{q_1,..,q_j\}}} f_t) \qquad \hat{V}_{\mathrm{non-pl}} = \sigma \sum_{\substack{t\in \\ \{q_1,..,q_{l-1}\}}} f_t \qquad \text{(A.6)}$$

In this case, the non-pipelined approach is more advantageous whenever $\hat{V}_{\mathrm{semi-pl}} > \theta\hat{V}_{\mathrm{non-pl}}$, where $\theta$ is a system-dependent performance tuning parameter. Further we reduce the number of system-dependent parameters by introducing $\alpha = \frac{\sigma}{\tau}\theta$ and modify Alg. A.1 with Alg. A.3 and Alg. A.2 with Alg. A.4. This modification allows a query to be executed in either a semi-pipelined or a non-pipelined way, depending on the data distribution and the choice of $\alpha$.

**Algorithm A.3**: Changes to processQuery($q$) line 4

```
1  V̂_semi-pl ← 0; V̂_non-pl ← 0; tmp ← 0;
2  foreach q_i ∈ q_1, .., q_l-1 do
3      foreach t_j ∈ q_i do
4          tmp ← tmp + f_t_j;
5          V̂_non-pl ← V̂_non-pipelined + f_t_j;
6      if tmp > L then
7          tmp ← L;
8      V̂_semi-pl ← V̂_semi-pl + tmp;
9  if V̂_semi-pl > α V̂_non-pl then
10     set the node with argmax_i(Σ_{t∈q_i} f_t) as the query processing node;
11     mark b as nobundle;
12 else
13     mark b as real;
```

**Algorithm A.4**: Append to processBundle($b$) line 14

```
1  else if b is marked as nobundle then
2      for t ∈ q_idx do fetch I_t;
3      if l > 1 then
4          put these into the b; mark the b as nobundle_res and send to the query processing
           node;
5      else
6          decompress the posting lists;
7          process the posting data; extract, sort and post-process the top-results; return these
           to the query broker;
8  else if b is marked as nobundle_res then
9      extract and decompress the posting lists from b;
10     if all the other posting lists are received and decompressed then
11         similar to line 7 of this algorithm;
```

### A.4.3 Alternative Routing Strategy

According to our last observation, the query route chosen by increasing $\min_{t\in q_i} F_t$ order is not always the optimal one. As each node is visited only once, the algorithm processes all

the terms stored on this node before processing any other term. Assume four query terms $t_1$, $t_2$, $t_3$ and $t_4$ with collection frequencies $F_{t_1} < F_{t_2} < F_{t_3} < F_{t_4}$ and three sub-queries $sq_1 = \{t_1, t_4\}$, $sq_2 = \{t_2\}$ and $sq_3 = \{t_3\}$. In this case the algorithm will first process $I_{t_1}$ and $I_{t_4}$, then process $I_{t_2}$ and finally $I_{t_3}$. Now, if $f_{t_4}$ is larger than $L$ it will fill up the rest of the accumulator data structure. The number of accumulators to be transferred from this node is expected to be close to $L$. If both $I_{t_2}$ and $I_{t_3}$ are shorter than $L$, none of these will trigger a re-adjustment of $h_t$ and $v_t$ values used by the pruning method (see Sec. A.3), and therefore at least $L$ elements will be transferred also between the last two nodes.

There are two solutions to this problem. The first one is to modify the pruning method to be able to delete accumulators even if the current list is shorter than $L$. However, this will lead to loss of already created accumulators that still might be present in the result set. Instead, we can look at an alternative routing strategy.

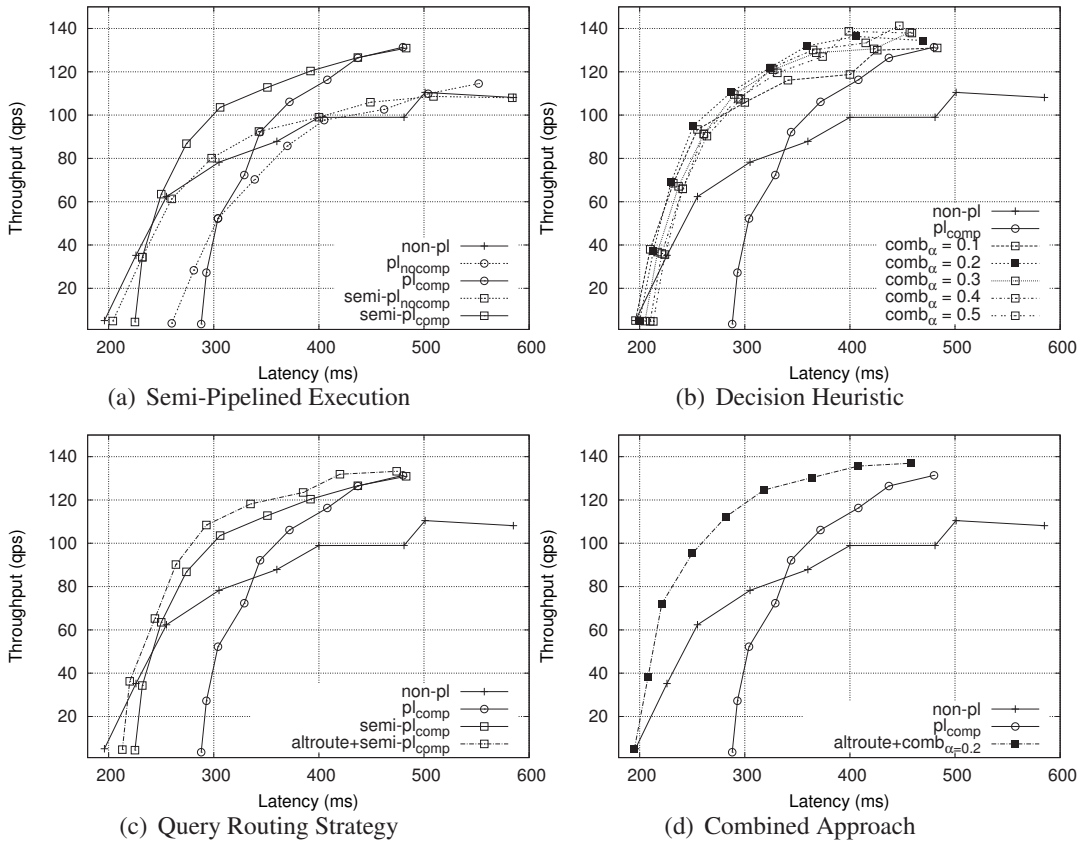
In order to minimize the network load we propose to route the query not according to the increasing smallest term frequency, but according to *increasing longest posting list length* by replacing $\min_{t \in q_i} F_t$ with $\max_{t \in q_i} f_t$ in Alg. A.1. In the next section we will show that this routing strategy incurs no significant reduction in the result quality, but does reduce the number of transferred accumulators and thus improves the average query latency and throughput.

## A.5 Evaluation

In order to evaluate our ideas and compare the final method to the state-of-the-art methods we have extended the Terrier Search Engine [14] to a distributed search engine. We use all the original index data structures with minor modifications to provide support for distributed and concurrent query processing. Further we index the 426GB TREC GOV2 corpus and distribute the index across a cluster of 8 nodes. Each of the nodes has two 2.0GHz Quad-Core CPUs, 9GB memory, one 16GB SATA disk and runs Linux kernel v2.6.33.2, and Java SE v1.6.0_16. The nodes are interconnected with a 1Gb network.

For the document collection, we apply both stemming and stop-word removal. We distribute posting lists according to $hash(t)$ mod number of nodes. For the performance experiments we use the first 20000 queries from the Terabyte Track 05 Efficiency Topics [4], where the first 10000 are used as a warm-up and the next 10000 as an evaluation set. We make sure to drop OS disk caches before each experiment and all the experiment tests are performed twice, where the best of the two runs is considered as the result.

Each of the nodes runs an instance of our search engine, consisting of a modified version of Terrier, a communication manager, and a pool of query processing tasks. Processing on each node is done by 8 concurrent threads. The number is chosen according to the number of processor cores. One of the nodes works also as the query broker. For each experiment we set a fixed maximum number of queries to be executed concurrently, we refer to this value as *concurrency level*. The concurrency level is varied between 1 and 64, with steps at 8, 16, 24, 32, etc. The upper bound is chosen to be 64 since the maximum number of queries executing concurrently on each of the 8 nodes is 8. This means that




(a) Semi-Pipelined Execution (b) Decision Heuristic

(c) Query Routing Strategy (d) Combined Approach

Figure A.2: Query latency and throughput with varied concurrency.

for a concurrency level larger than 64, queries will be queued on at least one of the nodes, even with perfect load balancing.

Finally, as the reference methods we use implementations of the non-pipelined and the pipeline approaches based on the pruning method by Lester et al.. For all of the experiments we use $L = 400000$. As the document weighting model we use the variant of Okapi BM25 provided by Terrier.

The results are organized as follows. In Figure A.2 we present a comparison of each of the proposed techniques, as well as the final combination, and compare these to the pipelined and non-pipelined approach. Each graph plots average query throughput against query latency at different concurrency levels. For every plot the lowest concurrency level (1) corresponds to the point with the shortest latency, and the highest concurrency level (64) corresponds to the point with the longest latency. Further details on throughput and latency for some of these methods are given in Table A.2. For each of the methods the table presents the shortest latency, shortest latency with throughput over 100 QPS, highest throughput and the best throughput/latency ratio. Finally, in Table A.1 we compare our routing strategy to the original one in terms of results quality and the number of transferred accumulators.

### Semi-Pipelined Execution

The plots presented in Figure A.2(a) compare the performance of the semi-pipelined approach ($semi-pl$) to pipelined ($pl$) and non-pipelined ($non-pl$). For the first two methods we also present the results with accumulator set compression applied (subscript $comp$) and without ($nocomp$). From the results the non-pipelined approach provides short query latency at low concurrency levels, but fails to achieve high throughput at higher concurrency levels as the nodes saturate on transfer, disk-access and posting list processing.

The pipelined approach with no compression has longer latencies to begin with, and achieves only a slightly better maximum throughput as the network and disk access become overloaded. With compression applied the method incurs even longer latencies at low concurrency levels, but succeeds to achieve a significantly higher throughput and shorter latency at higher concurrency levels. The improvement comes from a reduced network load.

The semi-pipelined approach with no compression performs quite similar to the non-pipelined approach, with a short query latency at lower concurrency levels, but not as good throughput at higher concurrency levels. However, with compression applied the method succeeds to achieve as high throughput as the pipelined approach, while it has a significantly shorter latency at lower concurrency levels. The method has also the best throughput/latency trade-off compared to the other methods, it succeeds to provide a throughput above 100 QPS with a query latency around 300 ms.

### Decision Heuristic

In Figure A.2(b) we plot the performance measurements of the decision heuristic ($comb_\alpha$) with different values of $\alpha$. The method combines the semi-pipelined approach with compression applied together with the non-pipelined approach. The values of $\alpha$ we try are between 0.1 and 0.7 with a step at 0.1. We re-plot the pipelined and non-pipelined approaches to provide a reference point.

The figure shows that for the values $\alpha = 0.4-0.6$ the method provides the highest throughput (around or above 140 QPS), while $\alpha = 0.1$ minimizes latency at lower concurrency levels. The best overall performance, in our opinion, is obtained by $\alpha = 0.2$, with over 110 QPS in throughput at less than 300 ms in query latency.

### Alternative Routing Strategy

In Figure A.2(c) we show the performance improvement of the semi-pipelined approach with the alternative routing strategy ($altroute + semi - pl$). From the results, the method improves both latency and throughput by reducing the number of transferred accumulators.

In Table A.1 we provide the results on the quality of results and the number of transferred accumulators. The results are obtained from the experiments with the TREC GOV2 corpus, the TREC Adhoc Retrieval Topics and Relevance Judgements 801-850 [3]. As

| Method | mean average precision | average recall | final number of accs. | number of transferred accs. |
|---|---|---|---|---|
| Full (1 Node) | 0.357 | 0.699 | 3130129 | N/A |
| Lester (1 Node) | 0.356 | 0.701 | 348268 | N/A |
| TP MinTF 8 Nodes | 0.356 | 0.701 | 347220 | 340419 |
| TP MinMaxDF 8 Nodes | 0.357 | 0.697 | 349430 | **318603** |
| TP MinTF 4 Nodes | 0.356 | 0.701 | 346782 | 272032 |
| TP MinMaxDF 4 Nodes | 0.356 | 0.700 | 350598 | **247620** |
| TP MinTF 2 Nodes | 0.356 | 0.701 | 354656 | 177166 |
| TP MinMaxDF 2 Nodes | 0.358 | 0.703 | 341732 | **165953** |

Table A.1: Effects of the routing method on the quality of results and average number of the final and transferred accumulators for `06topics.801-850` and `qrels.tb06.top50`, $L = 400000$.

the table shows, the alternative routing strategy does not reduce the quality of results, but reduces the number of transferred accumulators by 6.4 to 9.0%.

However, it is not the average but the worst case performance we are trying to improve. As an example, Topic 807 - `'Sugar tariff-rate quotas'`, scheduled by smallest collection frequency will process {'quota' $F_t = 186108$, $f_t = 44395$, 'rate' $F_t = 10568900$, $f_t = 1641852$} as the first sub-query, {'sugar' $F_t = 281569$, $f_t = 109253$} as the second, and {'tariff' $F_t = 513121$, $f_t = 80017$} as the third. The total number of transferred accumulators will be 693244. Using our routing strategy the sub-query ordering will be {'tariff'}{'sugar'}{'quota rate'}. As the result, the number of transferred accumulators will be 265396, 61.7% less than with the original method, while the MAP and recall will be the same.

### Combination of the Methods

In Figure A.2(d) we compare a final combination ($altroute + comb$) of the semi-pipelined approach, decision heuristic with $\alpha = 0.2$ and the alternative routing strategy against the state-of-the-art approaches. More details for this method as well as the main methods discussed so far are given in Table A.2. As the results show, our method outperforms both of the state-of-the-art method as it provides both a short query latency at lower concurrency levels and a high throughput at higher concurrency levels. Compared to the non-pipelined approach our method achieves up to 26% higher throughput and 22% shorter latency. Compared to the pipelined approach the method provides up to 47% higher throughput and up to 32% shorter latency. Our final method achieves also the best throughput/latency ratio, 112.2 QPS with 282ms in average response time per query. Finally, the best throughput value presented in Table A.2 at 137.0 QPS was achieved by the final combination with a corresponding latency 458 ms. From the Figure A.2(d), the method achieves 135.7 QSP with a corresponding latency at 408 ms, which can be considered as a quite good trade-off, 10.9% shorter latency for 0.9% lower throughput.

| Method | shortest latency | shortest latency 100QPS | highest throughput | best tp./lat. |
|---|---|---|---|---|
| $non - pl$ | **5.1/196** | 110.5/501 | 110.5/501 | 99.0/400 |
| $pl_{nocomp}$ | 3.8/260 | 102.6/462 | 114.5/552 | 97.7/405 |
| $pl_{comp}$ | 3.5/288 | **106.1/372** | **131.4/480** | **126.4/437** |
| $semi - pl_{nocomp}$ | 4.9/204 | 106.0/449 | 108.6/509 | 92.4/343 |
| $semi - pl_{comp}$ | 4.4/225 | 103.5/306 | 131.0/483 | 103.5/306 |
| $comb_{\alpha=0.2}$ | 5.0/200 | 110.5/287 | 136.4/406 | 110.5/287 |
| $altroute + semi - pl_{comp}$ | 4.7/213 | 108.4/293 | 133.3/474 | 108.4/293 |
| $altroute + comb_{\alpha=0.2}$ | **5.1/195** | **112.2/282** | **137.0/458** | **112.2/282** |

Table A.2: The most important query latency and throughput measurements.

## A.6 Conclusions and Further Work

In this paper we have presented an efficient alternative to the pipelined approach by Moffat et al. [13] and the ad-hoc non-pipelined approach. Our method combines non-pipelined disk-accesses, a heuristic method to choose between pipelined and non-pipelined posting list processing, and an efficient query routing strategy. According to the experimental result, our method provides a higher throughput than the pipelined approach, a shorter latency than the non-pipelined approach, and significantly improves the overall throughput/latency ratio.

Further improvements can be done by using a decision heuristic to choose wherever posting list fetch and decompression should be done beforehand, in a pipelined way, or just partially delayed. The decision depends on the number of queries executing concurrently, size of the posting list, size of the main memory or index access latency and decompression speed. The decision heuristic we use can be replaced by a more advanced one, based on an analytical performance model similar to the one presented by Lucchese et al. [6], which shall give a further improvement. The routing strategy we have presented is quite general and can be substituted by a more advanced approach that will account CPU and network load on different nodes or similar. Finally, from our results, compression is one of the main keys to achieve a high throughput and short query latencies at higher concurrency levels. However, there is a trade-off between the amount of work spent on compressing/decompressing the data and the actual benefit of it. Also not all queries/accumulator sets benefit from compression and, for example, at lower concurrency levels there might be no need to compress the data. These observations should be considered in the future.

## A.7 References

[1] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. SPIRE*, 2001.

[2] C. Badue, R. Barbosa, P. Golgher, B. Ribeiro-Neto, and N. Ziviani. Basic issues on the processing of web queries. In *Proc. SIGIR*. ACM Press, 2005.

[3] S. Büttcher, C. Clarke, and I. Soboroff. The trec 2006 terabyte track. In *Proc. TREC*, 2006.

[4] C. Clarke and I. Soboroff. The trec 2005 terabyte track. In *Proc. TREC*, 2005.

[5] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proc. WISE*, 2005.

[6] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Proc. InfoScale*, 2007.

[7] A. MacFarlane, J. McCann, and S. Robertson. Parallel search using partitioned inverted files. In *Proc. SPIRE*. IEEE Computer Society, 2000.

[8] M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *Proc. CIKM*. ACM, 2007.

[9] M. Marin, V. Gil-Costa, C. Bonacic, R. Baeza-Yates, and I. Scherson. Sync/Async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 36(4), 2010.

[10] M. Marin and C. Gomez. Load balancing distributed inverted files. In *Proc. WIDM*. ACM, 2007.

[11] M. Marin, C. Gomez-Pantoja, S. Gonzalez, and V. Gil-Costa. Scheduling intersection queries in term partitioned inverted files. In *Proc. Euro-Par*. Springer-Verlag, 2008.

[12] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proc. SIGIR*. ACM Press, 2006.

[13] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3), 2007.

[14] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *OSIR Workshop, SIGIR*, 2006.

[15] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proc. DL*. ACM Press, 1998.

[16] G. Zipf. Human behavior and the principle of least-effort. *Journal of the American Society for Information Science and Technology*, 1949.

[17] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Appendix B

# Paper B.I: Efficient Compressed Inverted Index Skipping for Disjunctive Text-Queries

Simon Jonassen and Svein Erik Bratsberg.
*Appeared at the 33rd European Conference on Information Retrieval (ECIR), Dublin, Ireland, April 2011.*

**Abstract:** In this paper we look at a combination of bulk-compression, partial query processing and skipping for document-ordered inverted indexes. We propose a new inverted index organization, and provide an updated version of the MaxScore method by Turtle and Flood and a skipping-adapted version of the space-limited adaptive pruning method by Lester et al. Both our methods significantly reduce the number of processed elements and reduce the average query latency by more than three times. Our experiments with a real implementation and a large document collection are valuable for a further research within inverted index skipping and query processing optimizations.

## B.1 Introduction

The large and continuously increasing size of document collections requires search engines to process more and more data for each single query. Even with up-to-date inverted index partitioning, distribution and load-balancing approaches the performance of a single node remains important.

A large number of methods aimed to reduce the amount of data to be fetched from disk or processed on CPU have been proposed. Among these we find static and dynamic pruning, impact-ordered lists, compression, caching and skipping. In this paper we look at document-ordered inverted lists with a combination of bulk-compression methods, partial query processing and skipping.

One of the main challenges associated with processing of disjunctive (OR) queries is that documents matching *any* of the query terms might be returned as a result. In contrast, conjunctive (AND) queries require to return only those documents that match *all* of the terms. In the latter case, the shortest posting list can be used to efficiently *skip* through the longer posting lists and thus reduce the amount of data to be processed. An approach proposed by Broder et al. [2] was therefore to process a query as an AND-query, and only if the number of results is too low, process it once again as the original query. Instead, we look at two heuristics, the *MaxScore* method by Turtle and Flood [15] and the space-limited adaptive pruning method by Lester et al. [10], with a purpose to apply skipping in a combination with OR-queries.

The recent publications by Suel et al. [8, 16, 17] have demonstrated superior efficiency of the PForDelta [19] and its variants compared to the alternative index compression methods. For 32 bit words PForDelta compresses data in chunks of 128 entries and does fast decompression of data with unrolled loops. However, to our knowledge, the only skipping alternative considered by Suel et al. was storing the first element of each *chunk* in main memory. On the other hand, the skipping methods published so far optimize the number of *entries* to be fetched and/or decompressed, with no consideration of disk buffering optimizations, internal CPU caches and bulk-decompression.

The main motivation behind this paper is to process disjunctive queries just as efficiently as conjunctive. We expect that a proper combination of inverted index compression, skipping and query optimization techniques is sufficient to do so. The contribution of our work is as follows. (a) We present a novel and efficient skipping organization designed specifically for a bulk-compressed disk-stored inverted index. (b) We revise the MaxScore-heuristics and present a complete matching algorithm. (c) We present a modification of the pruning method by Lester in order to enable skipping. (d) We evaluate the performance of the inverted index and skipping methods against state-of-the-art methods with the GOV2 document collection and a large TREC query set on a real implementation, and provide important experimental results. Our methods significantly improve query processing efficiency and remove the performance gap between disjunctive and conjunctive queries. Finally, we show that, due to disk-access overhead, skipping more data does not necessary reduce the query latency.

This paper is organized as follows. Section B.2 gives a short overview of related work. Section B.3 presents the structure of our compressed self-skipping index. Section B.4 revises the MaxScore method and presents an improved version of Lester's algorithm. The experimental framework and results are given in Section B.5. The final conclusions and directions for further work follow in Section B.6.

## B.2 Related Work

**Query optimizations.** Early query optimization strategies for inverted indexes have been considered by Buckley and Lewit [3]. Turtle and Flood [15] have discussed and evaluated a number of techniques to reduce query evaluation costs. We use `MaxScore` to refer to a document-at-a-time (DAAT) partial ranking optimization based on the maximum achievable score of a posting list and the score of the currently lowest ranked document mentioned in the paper.

However, the original description of MaxScore [15] omits some important details, and it differs from a later description by Strohman, Turtle and Croft [14]. Also the recited description of the method provided by Lacour et al. [9] is closer to the original rather than the later description of the method. We believe that both descriptions [14, 15] are correct and explain two different heuristics that can be combined. We find this combination to be highly efficient, but lacking a clear, unified explanation. For this reason we present a complete algorithmic implementation of MaxScore and explain how skipping can be done.

Moffat and Zobel [12] have presented the original *Quit/Continue* strategies for term-at-a-time processing (TAAT) and explained how these can be combined with efficient skipping. The paper also explains the choice of skipping distance for single- and multiple-level skipping. In a later comparison paper, Lacour et al. [9] have found these optimizations most efficient, while MaxScore was only slightly better than full TAAT evaluation. However, as we understand, the MaxScore implementation by Lacour et al. was limited only to partial ranking [15] with no skipping. We show that, compared to this method, skipping improves the average query latency with by a factor of 3.5.

Lester et al. [10] introduced an efficient space-limited TAAT query evaluation method, which provides a trade-off between the number of maintained *accumulators* (ie. partially scored candidates) and the result quality. However, no considerations of skipping have been made by the authors. In our work, we present a modification of the method that does inverted list skipping and demonstrate a significant performance improvement.

**Compression.** PForDelta compression was originally proposed by Zukowski [19]. Suel et al. [17] have demonstrated the efficiency of this method compared to the other methods and suggested a number of improvements [16]. Skipping has been mentioned in most of the PForDelta related papers by Suel et al., but the only implementation considered so far was to store the first document ID from each chunk of each posting list, uncompressed, in main memory [8]. Opposite to this, we suggest a self-skipping compressed inverted index.

**Skipping.** Moffat and Zobel [12] wrote one of the first papers applying inverted index skipping and presented a method to choose optimal skip-lengths for single- and multiple-

level skipping with respect to disk-access and decompression time. The model behind the method assumes to fetch and decompress an *element* at a time, and the optimization is done on the total number of fetched and decompressed entries. Instead, we assume data to be compressed in *chunks* and stored in *blocks* with its implications to the processing model.

Other methods to estimate optimal skipping distances were presented by Strohman and Croft [13], Chierichetti et al. [6] and Boldi and Vigna [1]. The first paper looks at skipping with impact-ordered inverted files and the second one looks at spaghetti skips in doctored dictionaries which are not related to our focus. The skipping structure described by Boldi and Vigna [1] has a certain similarity with ours. The smallest number of pointers to be skipped is a group of 32 or 64 entries, each compressed on its own, and skipping pointers are stored in towers. Our skipping structure, as we explain in the next section, compresses groups of 128 index postings or 128 skipping pointers in chunks, while the pointers corresponding to different *skipping-levels* are stored in different chunks.

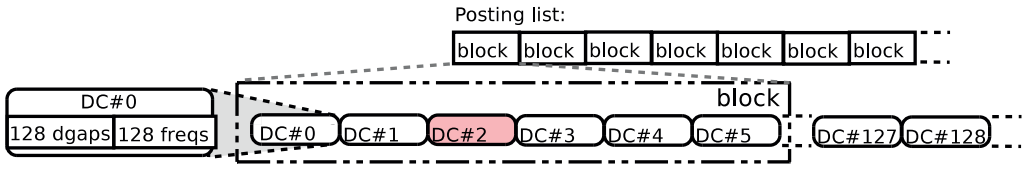
Büttcher and Clarke [4] have presented an efficient I/O optimized random-access structure for random inverted index access. While having an insignificant similarity with our skipping organization, their method operates with nodes of constant byte size, such as an L2 cache line or a memory page. Our skipping structure operates with nodes of 128 elements and we separate physical block-size (used for disk-fetching) from the index layout itself. Finally, we optimize query *processing* rather than random access, and we look at PForDelta compression which was considered by the authors only as promising further work.

## B.3 Index Organization and Skipping

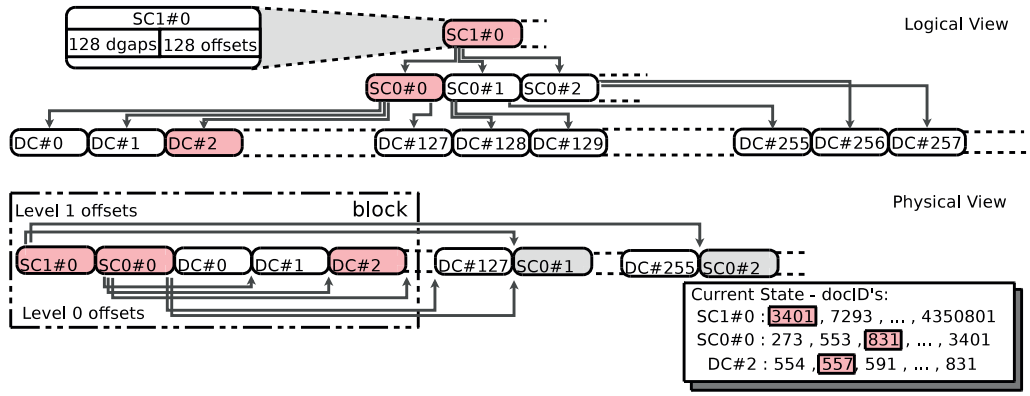
We look at processing of disjunctive queries with a disk-stored document-ordered inverted index [18]. Each posting list entry represents a document ID and a *term frequency* $f_{d,t}$ denoting the number of occurrences. Additionally to the inverted index that we describe below, we store a *lexicon file*, a *document dictionary* and additional statistics such as the total number of unique terms, documents, postings and tokens. The lexicon file stores a posting list pointer and the corresponding *collection frequency* $F_t$ and *document frequency* $f_t$ for each indexed term. The document dictionary stores mapping between document IDs, original document naming and document lengths.

### B.3.1 Basic Inverted Index

Without skipping, we split posting lists in chunks of 128 entries. Each chunk consists of 128 document IDs and 128 frequencies, where the last chunk contains a maximum of 128 entries. We use *d-gaps* instead of the original IDs. Further, each group of d-gaps and frequencies is compressed on its own, but using the same compression method. Chunks with more than 100 entries are compressed using NewPFD [16], a variant of PFoRDelta




(a) without skipping




(b) with skipping

Figure B.1: Compressed inverted index organization.

which stores highest order bytes of exceptions and exception offsets as two Simple9 encoded arrays. Chunks with less than 100 entries are VByte compressed. We illustrate this in Figure B.1(a).

Posting lists are processed using *iterators*. In this case, a list is fetched one *block* at a time. Blocks are zero-indexed, have a constant size $B$, but contain a varied number of chunks, and the block number $i$ has a start-offset of $B * i$ bytes. The choice of the block size itself is transparent from the physical organization. Once a block is fetched, chunks are decoded one at a time. All d-gaps and frequencies contained in a single chunk are decoded at once.

### B.3.2 Self-skipping Index

To support inverted index skipping we extract the last document ID and the *end-offset* from each chunk. These result in the lowest level of skipping hierarchy. Similar to the posting data itself, we divide these into groups of 128 elements, *skip-chunks*. The last document ID and the end-offset of each skip-chunk are recursively stored in a skip-chunk in the level above. As we illustrate in the upper part of Figure B.1(b), the logical structure reminds of a B-tree.

The physical layout of our self-skipping index resembles a prefix traverse of the logical tree. For this reason, the offsets stored in the lowest level skip-chunks represent the length of a corresponding *data-chunk*. For the levels above, an offset represents the length of a referred skip-chunk plus the sum of its offsets. This way, each chunk stores *real* offsets

between the chunks in a lower level, and the last offset corresponds to the offset to the next chunk at the same level. We illustrate this in the lower part of Figure B.1(b).

Document IDs within each skip-chunk are also gap-coded. Both d-gaps and offsets are compressed using the same compression methods as data-chunks, NewPFD for chunks with more than 100 elements and VByte otherwise. Additionally to using gaps instead of full IDs and relative instead of absolute offsets, NewPFD itself stores differences from a frame of reference, which rewards the regularity in document ID distribution and compression ratio. Finally, we avoid use of bit-level pointers, which improves both the index size and querying performance.

**Inverted Index Skipping.** As with the original index, we separate the choice of the block size from the index organization itself. We suggest to choose the size so that the first block of a list is likely to contain the first chunks of each skipping level and the first data chunk. Further we decompress each first skip-chunk from each level and the first data-chunk and use it as *state information* of the list iterator. We refer to these chunks as to *active* chunks (see Figure B.1(b)). We also convert d-gaps into document ID's and relative offsets into absolute *pointers*.

Now, a *skip($d$)*-operation can be done by comparing $d$ to the last document ID of the active data-chunk. If $d$ is greater, we compare $d$ to the last document IDs of the active skip-chunks from the lowest to the highest level. We proceed climbing up until we get a last document ID greater or equal $d$. For each chunk we mark also the entry corresponding to the currently active chunk level under. At this point we compare $d$ to the entries between the active and the last one until we get an entry with the document ID greater or equal $d$. Further, we fetch and decompress the chunk referred by the offset pointer. If the corresponding chunk is a data-chunk, we quickly find the required posting. Otherwise, we climb downwards until we get to a data-chunk. The worst-case number of decompressed chunks and fetched blocks in a random skip operation is therefore equal to the number of skip-levels, $O(log(F_t))$, where $F_t$ is the collection frequency of the term $t$.

## B.4 Query Processing Methods

Query processing is done by stemming and stop-word processing query tokens, looking-up the vocabulary, fetching and processing the corresponding posting list, followed by extraction, sorting and post-processing of the $K$ best results.

With term-at-a-time (TAAT) query processing, each posting list is processed at once, which allows certain speed-up, but requires to maintain a set of partial results, *accumulators*. Alternatively, with document-at-a-time (DAAT) query processing, all posting lists are processed in parallel and documents are scored one at a time. While DAAT processing has been considered more efficient in combination with skipping, the methods for term-partitioned distributed inverted files [11] apply TAAT processing. As we consider index skipping to be useful also for distributed query processing, we look at both methods.

**Algorithm B.1**: Skipping modification of the Lester's algorithm

**Data**: inverted index iterators $\{I_{t_1}, ...I_{t_q}\}$ sorted by ascending collection frequency $F_t$
**Result**: top $K$ query results sorted by descending score

```
1  A ← ∅; v_t ← 0.0; h_t ← 0;
2  foreach iterator I_t do
3      skipmode ← false;
4      if F_t < L then p ← F_t + 1;
5      else if v_t = 0.0 then calculate values of v_t, h_t, p and s according to Lester;
6      else
7          calculate new values of h_t and v_t from the old value of v_t;
8          if h_t ≥ f_max then p ← F_t + 1; skipmode ← true;
9      if skipmode then
10         foreach accumulator A_d ∈ A do
11             if I_t.docID < d then
12                 if I_t.skipTo(d) = false then A ← A − {A_* | A_* < v_t}; proceed to line 2;
13             if I_t.docID = d then A_d ← A_d + s(I_t);
14             if A_d < v_t then A ← A − A_d;
15     else
16         foreach document d ∈ I_t ⋃ A do
17             recalculate values of v_t, h_t, p and s when necessary;
18             if d ∉ I_t then
19                 if A_d < v_t then A ← A − A_d;
20             else if d ∉ A then
21                 if I_t.freq ≥ h_t then A_{I_t.docID} ← s(I_t); A ← A + A_{I_t.docID};
22             else
23                 A_d ← A_d + s(I_t);
24                 if A_d < v_t then A ← A − A_d;
25 return resHeap.decrSortResults(A);
```

### B.4.1 Term-At-A-Time Processing

The advantage of the Lester's method compared to the Continue approach was demonstrated in the original paper [10]. We prefer to look at the Lester's method instead of TAAT MaxScore [15] since the latter is a special case of the Continue approach. TAAT MaxScore stops creating new accumulators when the maximum achievable score of the next term falls below the current score of the $K$th candidate. Thus the method is equivalent to the Continue approach with an additional requirement to track the first $K$ top-scored candidates.

The modified version of the pruning method is given in Algorithm B.1. The algorithm is equivalent to the original [10] except from the lines 3, 8 and 9-15. The choice and usage semantics of the threshold variables $v_t$ and $h_t$ and adjustment variables $p$ and $t$ are explained in the original paper. $L$ is the target value for the number of maintained accumu-

**Algorithm B.2**: MaxScore

**Data**: inverted index iterators $\{I_{t_1}, ...I_{t_q}\}$ sorted by descending maximum score $\hat{s}(I_t)$
**Result**: top $K$ query results sorted by descending score

1 $q_{req} = q$; $resHeap \leftarrow \emptyset$;
2 calculate a set of cumulative maximum scores $\hat{a}$, $\hat{a}(I_{t_i}) = \sum_{i \leq j \leq q} \hat{s}(I_{t_j})$;
3 **while** $q > 0$ *and* $q_{\text{req}} > 0$ **do**
4  *score* $\leftarrow 0$; $d_{cand} \leftarrow min_{i \leq q_{\text{req}}}(I_{t_i}.doc)$;
5  **for** $i = 1; i \leq q; i \leftarrow i + 1$ **do**
6  **if** $score + \hat{a}(I_{t_i}) <$ *resHeap.minScore* **then** proceed to line 14;
7  **if** $i > q_{\text{req}}$ **then**
8  **if** $I_{t_i}$.*skipTo*$(d_{cand}) =$ *false* **then** remove $I_{t_i}$; update $\hat{a}$, $q$, $q_{\text{req}}$; **continue**;
9  **if** $I_{t_i}.doc = d_{cand}$ **then** *score* $\leftarrow$ *score* $+ s(I_{t_i})$;
10  **if** *score* $>$ *resHeap.minScore* **then**
11  *resHeap.insert*$(d_{cand}$, *score*$)$;
12  **for** $i = q_{\text{req}}; i > 1; i \leftarrow i - 1$ **do**
13  **if** $\hat{a}(I_{t_i}) <$ *resHeap.minScore* **then** $q_{\text{req}} \leftarrow q_{\text{req}} - 1$;
14  increment used and remove empty iterators $I_{t_{i \leq q_{\text{req}}}}$, update $\hat{a}$, $q$, $q_{\text{req}}$ if necessary;
15 return *resHeap.decrSortResults()*;

lators. Additionally, we introduce a constraint, $f_{\max}$, which can be chosen either statically or dynamically adjusted based on the maximum document frequency of the current term.

When $f_{max} < h_t$, the frequency required for a posting to be evaluated by the scoring function, $s(I_t)$, query processing switches into $skipmode$ (line 9-15), where no new accumulators will be created and therefore skipping can be performed. In skipmode, we use existing accumulators to skip through a posting list. With a low *accumulator target value* $L$, skipmode is achieved shortly after processing the first term and longer posting lists are efficiently skipped.

## B.4.2 Document-At-A-Time Processing

Our interpretation of MaxScore heuristics is given in Algorithm B.2. Prior to query processing we order the iterators by descending maximum score and calculate their cumulative maximum scores from last to first. Further, we say that terms from $t_1$ and up to $t_{q_{req}}$ are in the *requirement set*. We say that a term is in the requirement set if its cumulative maximum score is greater than the score of currently least ranked candidate, *resHeap.minScore*. From this definition, terms that are *not* in the requirement set can be skipped (line 7-9). Our algorithm begins with all terms in the requirement set, $q_{req} = q$. As the algorithm proceeds beyond the first $K$ documents, it reduces the requirement set (line 10-11) until there is only one term left. This idea is similar to the description given by Strohman, Turtle and Croft [14].

Each iteration of the algorithm begins by finding the lowest document ID within the requirement set (line 4). Further, we check every iterator from first to last and, if the

current score plus the cumulative maximum score of the remaining terms is less than *resHeap.minScore*, the algorithm stops further evaluation for this candidate and proceeds to the next one (line 6). This idea is similar to the description given by Turtle and Flood [15]. Finally, we terminate processing when the requirement set becomes empty or all postings have been processed.

## B.5 Evaluation

In order to evaluate index skipping and query processing algorithms we use the 426GB TREC GOV2 document corpus. We perform both stemming using the Snowball algorithm and stop-word removal on the document collection. The resulting index contains 15.4 million unique terms, 25.2 million documents, 4.7 billion pointers and 16.3 billion tokens. The total size of the compressed inverted index without skipping is 5.977GB. Additionally, we build another index with skipping, which adds 87.1MB to the index size, that is a 1.42% increase. The self-skipped inverted index contains 15.1 million posting lists with zero skip-levels, 279647 posting lists with one skip level, 15201 with two levels and only 377 with three levels.

For result quality evaluation we use the TREC Adhoc Retrieval Topics and Relevance Judgements 801-850 [5] without any modifications. For performance evaluation we use the Terabyte Track 05 Efficiency Topics [7]. As we look at optimizations for multi-keyword queries we remove any query with less than two terms matching in the index lexicon. From the original 50000 queries (having query length of 1 to 18 terms and an average length of 2.79 terms; matching 1-10, avg. 2.41 terms in the inverted index) we get 37132 queries (2-18, avg. 3.35; matching 2-10, avg. 2.91 terms), from which we extract a subset of the first 10000 queries.

All algorithms and data structures were implemented in Java. All the experiments were executed on a single workstation having an Intel Core 2 Quad 2.66GHz processor, 8GB RAM and a 1TB 7200RPM SATA2 hard-drive and running GNU/Linux. No caching optimizations have been done, OS disk cache was dropped before each run. For all experiments except the block size evaluation itself we operate with 16KB blocks. The query model used is the Okapi BM-25.

### B.5.1 Term-at-a-Time Processing

We compare our modification of the Lester's method using self-skipping index to the original method and full OR evaluation using a non-skipping index. Additionally, we compare it also to a self-skipping implementation of AND-processing. Our implementation of the AND method uses shortest posting list to skip through the longer ones. Any entries not matching in the later posting lists are removed from the accumulator set. Our implementation of the Lester's method is similar to the one used in the Zettair Search Engine[1]. It also avoids recalculating threshold variables if $h_t \geq f_{max}$ at the beginning of a posting

[1] `http://www.seg.rmit.edu.au/zettair/`

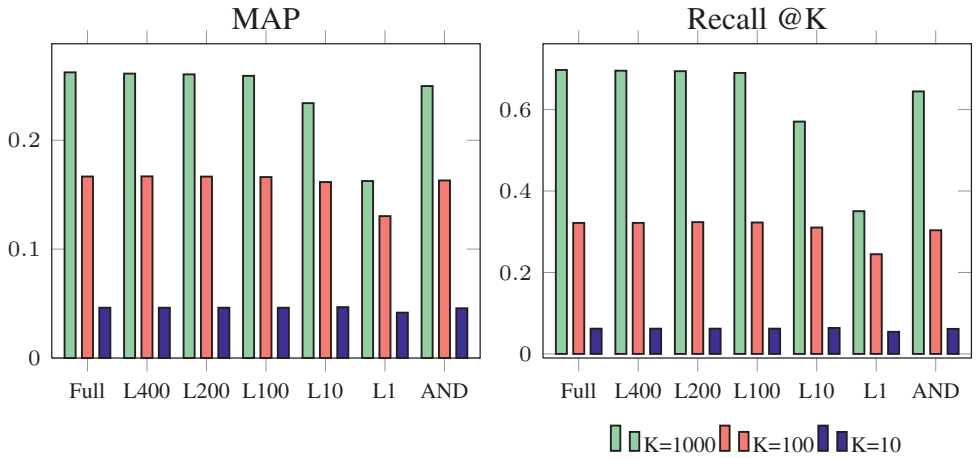



Figure B.2: Mean average precision and average recall comparison of the TAAT methods.

list. For both the Lester's method itself and the modification we use a statically defined $f_{max} = 2000$.

The result set with our modification is always equivalent to the one returned by the Lester's method itself. In Figure B.2 we illustrate the average mean of precision scores after each relevant document (MAP) and the total recall for the query retrieval topics. We evaluate Lester's method with accumulator set target values ($L$) 400000 (L400), 200000 (L200), 100000 (L100), 10000 (L10) and 1000 (L1). The results show that there is no significant difference in the result quality as long the result set size ($K$) is large enough. For $K = 1000$, the target value $L$ can be chosen as low as 100000 without a significant impact on the result quality. For $K = 100$ and 10, $L$ can be chosen as low as 10000.

The performance evaluation results are given in Figure B.3. For each method configuration we execute two runs. The first one, illustrated in the upper half of the figure, measures the total number of candidates inserted into the result heap, calls to the scoring function, frequencies and document IDs evaluated, decompressed chunks and fetched blocks. The second one, illustrated in the lower half, measures the resulting query latency (without profiling overhead). As the results show, our method significantly reduces the number of evaluated document IDs and frequencies compared to the other methods. The number of candidates inserted into the result heap and posting scorings is reduced similar to the original method. We observe also a decrease in the number of decompressed chunks due to inverted index skipping. However, the number of fetched blocks with a target value larger than 10000 is actually larger than with the other methods, due to storage overhead from skip-chunks. A significant reduction in the number of fetched blocks is observed only for $L = 1000$ (SL1), but as we will demonstrate in Section B.5.3, with a smaller block size, the number of fetched blocks will be reduced also for larger target values. Finally, our results show that the size of the result set $K$ influences only the number of heap inserts and does not affect any other counts nor the resulting query latency.

The measured average query latency is illustrated in the lower half of Figure B.3. For $L = 1000$, our modification is 25% faster than the original method and 69%, or more than three times, faster than a full evaluation. For a lower target value the performance of our

method is comparable to the AND-only evaluation. With $K = 100$ and $L = 10000$ our method outperforms the AND method, while the result quality is not affected.

### B.5.2 Document-at-a-Time Processing

The performance evaluation results of the DAAT methods are illustrated in Figure B.4. In these experiments we compare the full and partial evaluation methods without skipping, our interpretation of the MaxScore method with skipping, and AND and Partial AND methods with skipping. The partial evaluation method is similar to the MaxScore interpretation by Lacour et al. [9], the method avoids scoring the rest of the postings for a document if the current score plus the cumulative maximum score of the remaining terms falls below the score of the currently least ranked candidate. Partial AND (P.AND) combines skipping AND and partial evaluation. As the result set returned by the MaxScore method is identical to the full evaluation, we do not provide result quality evaluation for the DAAT methods.

Our result shows that the partial evaluation alone can significantly reduce the number of posting scorings and number of evaluated frequencies, which results in a 25% reduction in the average query latency. Our interpretation of MaxScore provides a significant further reduction to the number of evaluated document IDs. For $K = 1000$ it halves the number of decompressed chunks and reduces the number of fetched blocks. The average query latency with the MaxScore is 3.7 times shorter than with a full evaluation, or 2.8 times compared to the partial evaluation method alone. With $K$ less than 1000, the reduction in the number of decompressed chunks and fetched blocks is even more significant, and the performance of our method is close to the full AND (less than 31% difference). The improvement can be explained by increased $resHeap.minScore$, which allows to skip more data, and decreased overhead from changes in the candidate result set during processing.

### B.5.3 Physical Block Size

Figure B.5 illustrates the average latency versus fetched data volume per query for the first 1000 queries in the query log. The average number of fetched blocks can be obtained by dividing the data volume by the block size. As the results show, a small block size reduces the total data volume, most significantly for the MaxScore. However, the total latency improves gradually as the block size becomes larger. Decrease in the measured latency between 1 and 64KB blocks is 8.6% for Full DAAT, 9.8% for Full TAAT, 12.1% for skipping-Lester with $L = 100000$ and 13.7% for MaxScore. This can be explained by a decrease of random disk-seeks and the fact that, when a chunk is split between two blocks, it requires two block fetches to fetch the chunk. While not illustrated, the number of fetched blocks decreases as the block size increases. In summary, our results show that, without caching, small block sizes reduce only the total data volume, but not the resulting query latency.

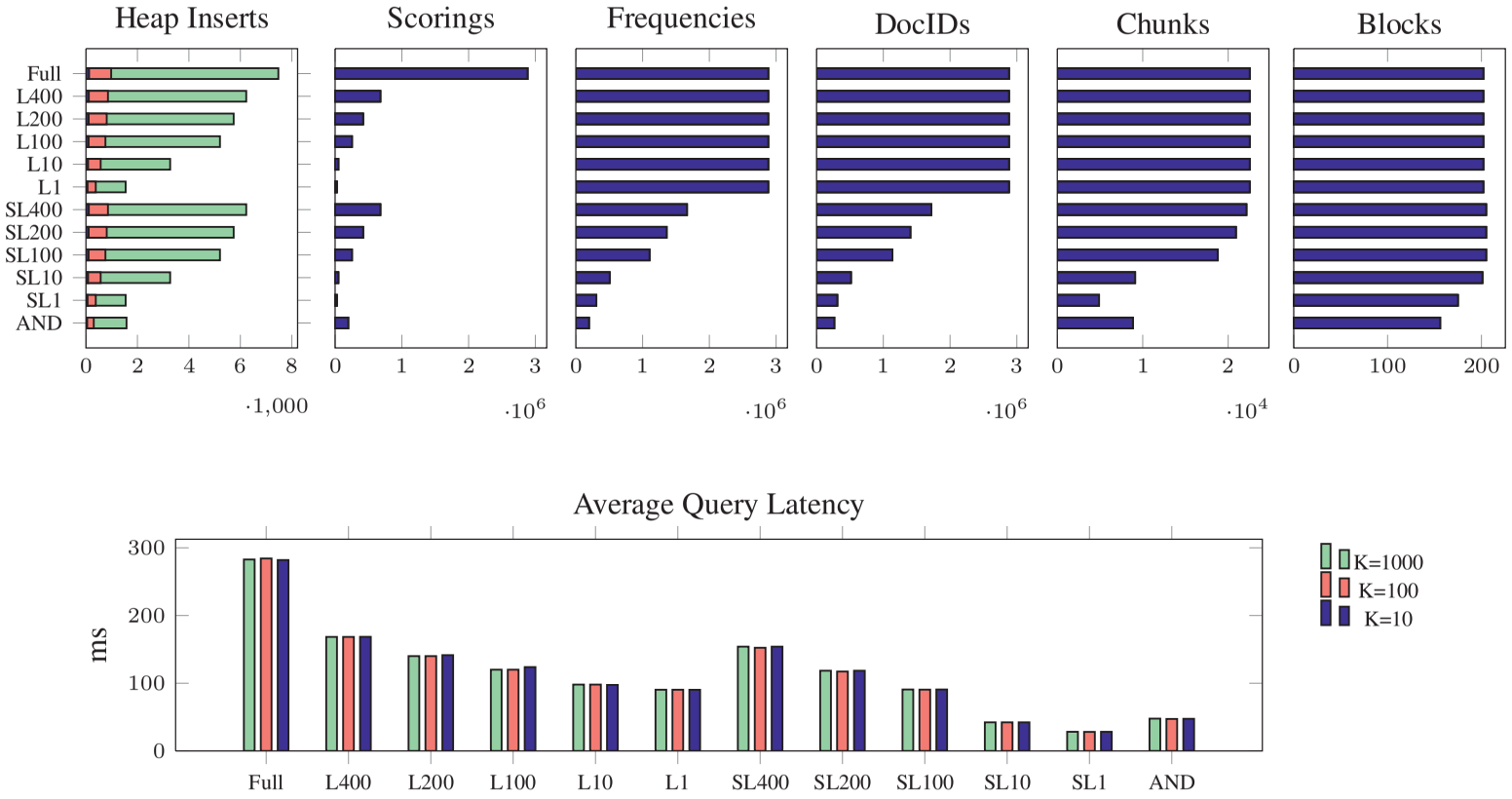



Figure B.3: Average number of processed entities (top) and query latency (bottom) of the TAAT methods. Our method is marked with "SL" (skipping-Lester).

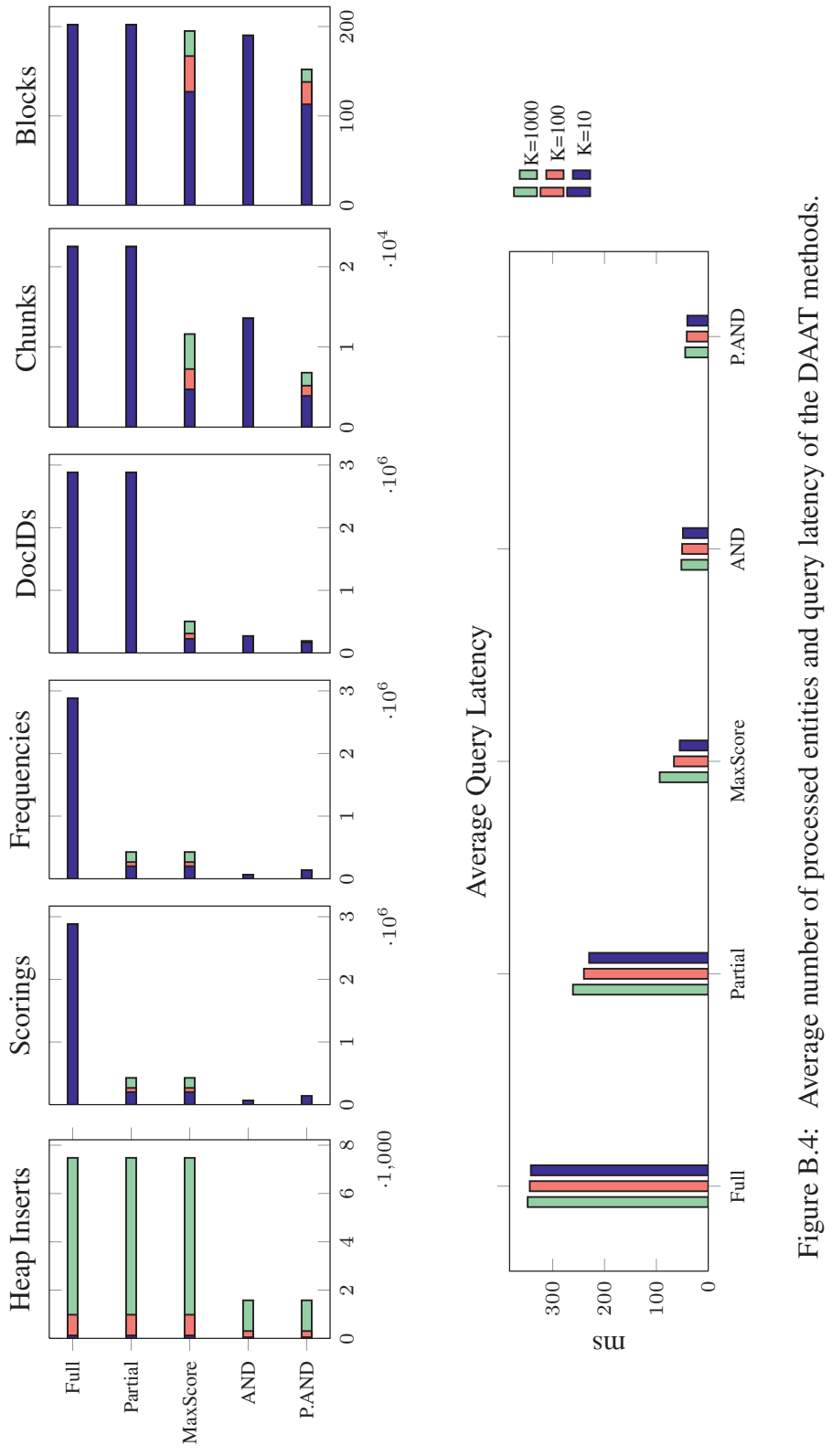



Figure B.4: Average number of processed entities and query latency of the DAAT methods.

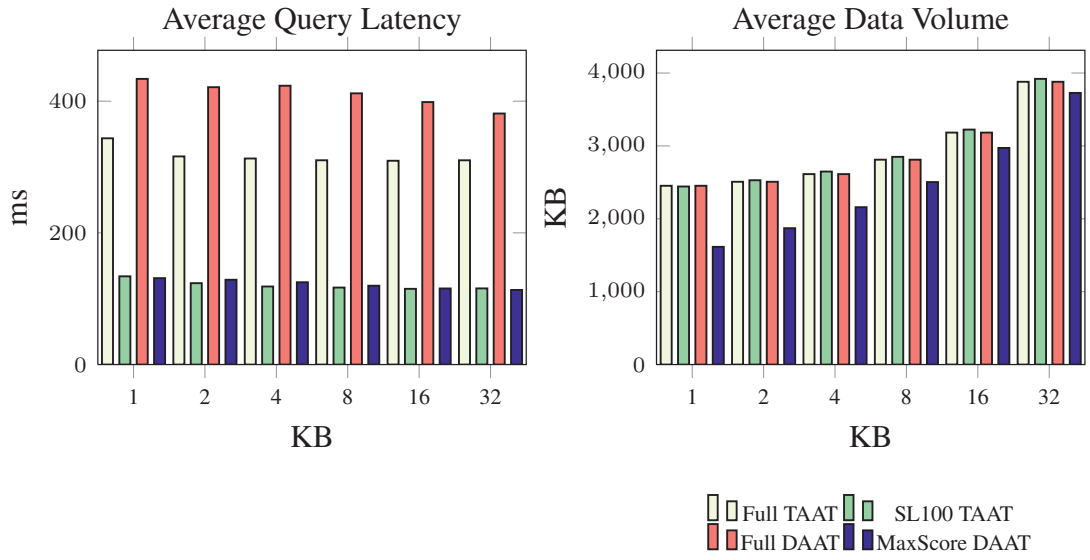



Figure B.5: Average query latency and data volume with varied block size, $K = 1000$.

## B.6 Conclusions and Further Work

In this paper we have presented an efficient self-skipping organization for a bulk-compressed document-ordered inverted index. From the experimental results, our query processing optimizations achieve more than three times speed-up compared to a full, non-skipping, evaluation and remove the performance gap between OR and AND queries. Further improvements can be done by postponed/lazy decompression of posting frequencies, extension of compression methods to 64 bit words and chunk-wise caching of posting lists.

**Acknowledgment:** This work was supported by the iAd Project funded by the Research Council of Norway and the Norwegian University of Science and Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the funding agencies.

## B.7 References

[1] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proc. SPIRE*, pages 25–28. Springer-Verlag, 2005.

[2] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*. ACM, 2003.

[3] C. Buckley and A. Lewit. Optimization of inverted vector searches. In *Proc. SIGIR*, pages 97–110. ACM, 1985.

[4] S. Büttcher and C. Clarke. Index compression is good, especially for random access. In *Proc. CIKM*, pages 761–770. ACM, 2007.

[5] S. Büttcher, C. Clarke, and I. Soboroff. The trec 2006 terabyte track. In *Proc. TREC*, 2006.

[6] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *Proc. WSDM*, pages 15–24. ACM, 2008.

[7] C. Clarke and I. Soboroff. The trec 2005 terabyte track. In *Proc. TREC*, 2005.

[8] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high-performance ir query processing. In *Proc. WWW*, pages 1213–1214. ACM, 2008.

[9] P. Lacour, C. Macdonald, and I. Ounis. Efficiency comparison of document matching techniques. In *Proc. ECIR*, 2008.

[10] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proc. WISE*, pages 470–477, 2005.

[11] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[12] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[13] T. Strohman and W. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, pages 175–182. ACM, 2007.

[14] T. Strohman, H. Turtle, and W. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, pages 219–225. ACM, 2005.

[15] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.

[16] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410. ACM, 2009.

[17] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, pages 387–396. ACM, 2008.

[18] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[19] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. ICDE*, pages 59–. IEEE Computer Society, 2006.

Appendix C

# Paper A.III: Improving the Performance of Pipelined Query Processing with Skipping

Simon Jonassen and Svein Erik Bratsberg.
*Appeared at the 13th International Conference on Web Information Systems Engineering (WISE), Paphos, Cyprus, November 2012.*

**Abstract:** Web search engines need to provide high throughput and short query latency. Recent results show that pipelined query processing over a term-wise partitioned inverted index may have superior throughput. However, the query processing latency and scalability with respect to the collections size are the main challenges associated with this method. In this paper, we evaluate the effect of inverted index skipping on the performance of pipelined query processing. Further, we introduce a novel idea of using Max-Score pruning within pipelined query processing and a new term assignment heuristic, partitioning by Max-Score. Our current results indicate a significant improvement over the state-of-the-art approach and lead to several further optimizations, which include dynamic load balancing, intra-query concurrent processing and a hybrid combination between pipelined and non-pipelined execution.

# C.1 Introduction

Two fundamental index partitioning methods, *term-wise* and *document-wise*, have been extensively compared during the last 20 years. The decision about which method gives the best performance relies on a number of system aspects such as query processing model, collection size, pruning techniques, query load, network and disk-access characteristics [3]. Rather than asking whether term-wise partitioning is generally more efficient than document-wise or not, we look at *pipelined query processing* [10] over a term-wise partitioned index and try to resolve the greatest challenges of the method. The advantage of term-wise partitioning is that a query containing $l$ query terms involves only $l$ processing nodes in the worst case. At the same time, it has to process only $l$ posting lists in total. Whether $l$ is the number of lexicon lookups or the total number of disk-accesses depends on the implementation. Pipelined query processing consists of routing a query bundle through the nodes responsible for any of the terms appearing in a particular query, modifying the accumulator set at each of these nodes and finally extracting the results at the last node. Compared to a traditional query processing approach, where each of the nodes retrieves the posting lists and the query processing itself is done solely by a ranker node, this method improves the workload distribution across the nodes and the overall performance [10].

In a recent book, Büttcher et al. [1] enlist three problems of pipelined query processing: poor scalability with increasing collections size, poor load balancing and limited intra-query concurrency due to term-at-a-time/node-at-a-time processing. In this work, we mainly address the first problem, but also show how our solution can be used to solve the other two.

Our contributions in this work are as follows. First, we explore the idea of combining inverted index skipping and pipelined query processing for a term-wise partitioned inverted index. Herein, we present an efficient framework and a skipping optimization to the state-of-the-art approach. Second, we present a novel combination of pipelined query processing, Max-Score pruning and document-at-a-time sub-query processing. Third, we present an alternative posting-list assignment strategy, which improves the efficiency of the Max-Score method. Fourth, we evaluate our methods with a real implementation using the TREC GOV2 document collection, two large query sets and 9 processing nodes and suggest a number of further optimizations.

The remainder of this paper is organized as follows. We briefly review the related work in Section C.2 and describe our framework in Section C.3. We present our query processing optimizations in Sections C.4 and C.5 and our posting list assignment optimization in Section C.6. We summarize our experiments in Section C.7. In Section C.8, we finally conclude the paper and suggest further improvements.

# C.2 Related work

Pipelined query processing was originally presented by Moffat et al. [10] as a method that retains the advantage of term-wise partitioning and additionally reduces the overhead at

the ranker node. However, due to a high load imbalance, the method was shown to be less efficient and less scalable than document-wise partitioning. In the following work by Moffat et al. [9] and Webber [13], the load balancing was improved by assigning posting lists in a fill-smallest fashion according to the workload (posting list size $\times$ past query frequency) associated with each term. The authors suggested also to multi-replicate the posting lists with the highest workloads and allow the broker to chose the replica dependent on the load. Additionally, Webber [13] has presented a strategy where each node may ask other nodes to report their current load in order to choose the next node in the route. The results reported by Moffat et al. [9] showed a significant improvement over the original (term-wise) approach, but not the document-wise approach. Under extended evaluation with a reduced main memory size performed by Webber [13] pipelined query processing outperformed document-wise partitioning in terms of maximum throughput. However, document-wise partitioning demonstrated shorter latency at low multiprogramming levels.

In a recent publication [4], we have addressed the problem of the increased query latency due to a strict node-at-a-time execution and presented a semi-pipelined approach, which combines parallel disk access and decompression and pipelined evaluation. Additionally, we suggested to switch between semi- and non-pipelined processing dependent on the estimated network cost and an alternative query routing strategy. While the results of this work indicate an ultimate trade-off between the two methods, it has several important limitations. First, similar to the original description [13], it requires to fetch and decompress posting lists completely. This minimizes the number of disk-accesses, but leaves a large memory footprint and endangers scalability of the method. Second, while the original approach fetches only one of a query's posting lists at a time, the semi-pipelined approach fetches all of them and keeps them in main memory until the query has been processed up-to and including this node. This improves the query latency, but increases the memory usage even further. Third, the compression methods used by the index may be considered outdated.

In the current work, we follow an alternative direction and try to improve pipelined query processing by means of skipping optimizations. In a recent work [5], we have investigated these techniques for a non-distributed index. In particular, we have presented a self-skipping inverted index designed specifically for NewPFoR [14]. Further, we have presented a complete description of document-at-a-time Max-Score. The Max-Score heuristic was originally presented by Turtle and Flood [12] without specifying enough details for skipping itself and later a very different description was given by Strohman et al. [11]. Our algorithm combines the advantage of both previous descriptions. Finally, we have presented a skipping version of the space-limited pruning algorithm by Lester et al. [7], which is the method used in the original description of pipelined query processing.

As a part of the current work, in Section C.8, we outline several further extensions. One of these, intra-query concurrent pipelined processing, has already been evaluated in our recent work [6]. The preliminary baseline of [6] is an early version of the Max-Score optimization we are about to present, $\text{MSD}_{\text{S}}^{\star}$, and the results of [6] are therefore directly applicable to this method.

In contrast to the previous work, we use skipping to resolve the limitations of distributed, pipelined query processing. Herein, we present several skipping optimizations and a new

term assignment strategy. In contrast to the previously presented assignment optimizations [2, 8, 15], our strategy does not try to assign co-occurring terms to the same node or to do load balancing, but rather to maximize the pruning efficiency. Additionally, it opens a possibility for dynamic load balancing with low repartitioning overhead and hybrid query processing. Finally, different from [4, 5, 6], our experiments use two query logs with very different characteristics, and a varied collection size.

## C.3 Preliminaries

For a given textual query $q$, we look at the problem of finding the $k$ best documents according to a similarity score $sim(d, q) = \sum_{t \in q} s(t, d)$, where $s$ is a function, such as Okapi BM25, estimating the relevance of a document $d$ to a term $t$. For this reason, we look at document-ordered inverted lists. Each list $I_t$ stores an ordered sequence of document IDs where the term $t$ appears and an associated number of occurrences of the term in the document $f_{t,d}$.

Our search engine runs on $n+1$ processing nodes, one of which works as a query broker and the remaining $n$ work as query processing nodes. For each of the indexed collection terms we build an inverted list and assign it to one of the processing nodes. By default, we use a hash-based term assignment strategy. Additionally to the inverted index, each query processing node maintains a small lexicon (storing term IDs, inverted file pointers, document and collection frequencies), a small replica of the document dictionary (storing the number of tokens contained in each document) and both partition and collections statistics. The query broker node stores a full document dictionary (document IDs, names and lengths), lexicon (tokens, term IDs, collection and document frequencies, IDs of the nodes a term is assigned to) and collection statistics. At runtime, the inverted index resides on disk and the remaining structures are in main memory.

The broker is responsible for receiving queries, doing all necessary preprocessing and issuing query bundles. The broker uses its lexicon to look-up query terms and to partition the query into several sub-queries consisting of the terms assigned to the same node. Further, it calculates a query route (i.e., a particular order to visit the query nodes) and creates a query bundle. The bundle contains term IDs, query frequencies and an initially empty document-ordered accumulator set. Accumulators represent document IDs and partial scores. Finally, the broker sends the query bundle to the first node in the route.

When a node receives a bundle, it decompresses its content and starts a new query processing task. First, it uses its own lexicon to find the placement of the posting list. In the next step, the node processes its own posting data with respect to the received accumulators and creates a new accumulator set. Then, it updates the query bundle with the new accumulator set and transfers it to the next node. Alternatively, the last node in the route extracts the top-$k$ results, sorts them by descending score and returns them to the broker.

**Algorithm C.1**: processBundle($b_q$) with skip-optimized space-limited pruning

**Data**: iterators $\{i_t\}$ and lexicon entries $\{l_t\}$ for $t \in q_j$ sorted by ascending $F_t$, accumulator set $A$, bundle attribute $v$, system parameters *avg_dl*, *use_opt*, $L$, $h_{\max}$

1 **foreach** $t \in q_j$ **do**
2 $A' \leftarrow A$, $A \leftarrow \emptyset$, *skipmode* $\leftarrow$ *false*, $p \leftarrow \lceil f_t / L \rceil$;
3 **if** $f_t < L$ **then**
4 $p \leftarrow f_t + 1$, $h \leftarrow 0$;
5 **else if** $v = 0$ **then**
6 set $h$ to the maximum of the first $p$ frequencies retrieved from $i_t$, $v \leftarrow s(l_t, h, avg\_dl)$, $i_t.reset()$;
7 **else**
8 **if** *use_opt* = *true and* $s(l_t, h_{\max}, avg\_dl) < v$ **then** *skipmode* $\leftarrow$ *true*;
9 **else** find $h \in [1, h_{\max}]$ s.t. $s(l_t, h, avg\_dl) \geq v$;
10 **if** *skipmode* = *false* **then**
11 $s \leftarrow \max(1, \lfloor (h+1)/2 \rfloor)$, $size_0 \leftarrow |A'|$; merge $A'$ and candidates from $i_t$ into $A$: calculate $i_t.s()$ only when $i_t.d() \in A'$ or $i_t.f() \geq h$, prune candidates having $s < v$; Each time $i_t.pos() = p$: $pred \leftarrow |A| + |A'| + (f_t - p) \times (|A| + |A'| - size_0)/p$, **if** $pred > 1.2 \times L$ **then** $h \leftarrow h + s$ **else if** $pred < L/1.2$ **then** $h \leftarrow \max(0, h - s)$ **endif**, $v \leftarrow s(l_t, h, avg\_dl)$, $s \leftarrow \lfloor (s+1)/2 \rfloor$, $p \leftarrow 2 \times p$;
12 **else**
13 **foreach** accumulator $\langle d', s \rangle \in A'$ **do**
14 **if** $i_t.d() < d'$ **then if** $i_t.skipTo(d') =$ *false* **then** add remaining accumulators s.t. $s \geq v$ to $A$, proceed to the next term (line 1);
15 **if** $i_t.d() = d'$ **then** $s \leftarrow s + i_t.s()$;
16 **if** $s \geq v$ **then** add $\langle d', s \rangle$ to $A$;
17 **if** it is the last node in the route **then** use a min-heap to find the $k$-best candidates from $A$, sort and return them to the broker **else** update $b_q$ with $A$ and $v$ and send it to the next node;

## C.4 Improving pipelined query processing with skipping

In this section, we describe the optimization to the state-of-the-art query processing approach. With this approach, the terms within each query are ordered by increasing collection frequency $F_t$ and the query itself is routed by increasing minimum $F_t$. Once a bundle is received by a node, the node extracts the accumulators and initiates posting list iterators. We use Algorithm C.1 to describe the following processing step performed by the query processor. The space-limited pruning method itself has been originally presented by Lester et al. [7] and the skipping optimization for a non-distributed index has already been presented in our previous work [5]. Therefore, the goal of this section is to describe the improvements to pipelined query processing. However, for the sake of intelligibility we also explain the most important details derived from the prior work [5, 7].

In the following, $i_t$ denotes the iterator of the posting list $I_t$, which provides methods to get the document ID, frequency, score and position of the current posting, advance to the next posting or to the first posting having $d \geq d'$ (both $next()$ and *skipTo*($d'$) return *false* if the end has been reached), or reset to the beginning. Further, $f_t$ denotes the document

frequency of $t$ (i.e., number of documents) and $F_t$ - its collection frequency (i.e., number of occurrences), and $|A|$ - the current size of the accumulator set $A$.

The idea behind the pruning method (lines 1-7, 9, 11) is to restrict the accumulator set to a target size $L$. As the algorithm shows, the posting lists of a particular sub-query $q_j \subseteq q$ are evaluated term-at-a-time. As long the document frequency $f_t$ of the current term is below $L$, each posting is scored and merged with the existing accumulator set. Otherwise, the algorithm estimates a frequency threshold $h$ and a score threshold $v$, just as suggested by Lester et al. For this reason, $h$ is set to the maximum frequency among the first $p = \lceil f_t/L \rceil$ postings. The rationale here is that, if one out of $p$ postings will pass the frequency filter $f_{t,d} \geq h$, the total number of such postings will be $L$. Further, in order to prune existing accumulators, $v$ is calculated from $h$. Differently from Lester et al., our score computation uses also the length of a document. For this reason we calculate the threshold score using $l_t$, $h$ and the average document length *avg_dl*. Now, the algorithm is able to prune the existing accumulators having score $s < v$ and avoid scoring postings having $f_{t,d} < h$ and not matching in among the existing accumulators. Each time $p$ postings of $I_t$ has been processed, it predicts the size of the resulting accumulator set. Then, it either increases or decreases the frequency threshold, updates the score threshold and finally cools-down the threshold variation. Finally, if the thresholds have already been defined, it uses the previously computed $v$ to find the corresponding value $h$. Similar to the implementation in Zettair[1], we try only the values between 1 and a system-specific maximum value $h_{\max}$. Additionally, we apply binary search to reduce the number of score computations.

Our optimization (lines 8, 10, 12-16) suggests to switch the query processing into a conjunctive skip-mode when $s(l_t, h_{\max}, avg\_dl)$ is below the previously computed $v$. In this mode, a posting is scored only when there is already an existing accumulator with the same document ID. For the first sub-query, $b_q$ is initiated with $v = 0$. After processing a sub-query $q_j$, $b_q$ is updated with the current $A$ and $v$ and forwarded to the next node. This means that each query starts in the normal, disjunctive mode. When the optimization constraint holds, it switches into the conjunctive mode. However, if the next posting list is shorter than $L$, the processing will switch back to the normal mode, but it will proceed to prune the accumulators having $s < v$.

The benefit of our optimization depends also on the inverted index implementation. We apply the layout described in our previous work [5]. With the basic index, each posting list is divided into groups of 128 entries, which are stored as two chunks containing 128 document ID deltas and 128 frequencies, both compressed with NewPFoR. To support skipping, we build a hierarchy of skip-pointers, which consist of an end-document ID and an end-offset pointer to a chunk level below. Further, we calculate deltas and compress these in chunks of 128 entries using NewPFoR. Next, we prefix-traverse the logical tree while writing to disk in order to minimize the size of skip-pointers and optimize reading. At the processing time, each posting list iterator maintains one chunk from each skip-level decompressed in main memory and applies buffering while reading from disk. Different from the previous work, we decompress frequency chunks only when at least one of the corresponding frequencies has been requested.

[1] `http://www.seg.rmit.edu.au/zettair/`

With the basic index, the cost of a skip is proportional to the number of blocks (I/O) and chunks (decompression) between the two positions. With the self-skipping index, the operation is done climbing the logical hierarchy up (using already decompressed data) and down (reading and decompressing new data when necessary). Therefore, the upper bound cost of the operation in the number of decompressed chunks and read blocks is $O(\log(D))$.

## C.5 Max-Score optimization of pipelined query processing

The main drawback of the query processing methods discussed in the previous section lies in the unsafe pruning strategy. Additionally, these techniques are limited to term-at-a-time and node-at-a-time query processing. For this reason, we suggest an alternative query processing approach employing document-at-a-time processing of sub-queries, and later we show how this new method can be extended to provide intra-query parallelism. In order to guarantee safe pruning, we look at the Max-Score heuristic [5, 11, 12]. To give a better explanation, first we describe the idea for a non-distributed scenario, $q_j = q$, then how it can be applied to pipelined query processing, and finally, present the algorithm.

At indexing time, we pre-compute an upper-bound score $\hat{s}_t$ for each posting list $I_t$ (i.e., the maximum score that can be achieved by any posting). At query processing time, we order $q_j = \{t_1, \ldots, t_l\}$ by decreasing $\hat{s}_t$ and use $a_i$ to denote the maximum score of terms $\{t_i, \ldots, t_l\}$, i.e., $a_i = \sum \hat{s}_t$ s.t $t \in \{t_i, \ldots, t_l\}$. Further, $q_j$ is processed document-at-a-time and each iteration of the algorithm selects a new candidate, accumulates its score and eventually inserts it into a $k$-entry min-heap. The idea behind Max-Score is to prune the candidates that cannot enter the heap. As terms are always processed in order $t_1$ to $t_l$, they can be viewed as two subsets $\{t_1, \ldots, t_r\}$ (required) and $\{t_{r+1}, \ldots, t_l\}$ (optional), where $r$ is the smallest integer such that $a_r \geq \tilde{s}$ and $\tilde{s}$ is the current $k$-th best score. It is easy to see that candidates that do not match any of the required terms cannot enter the heap. Therefore, the candidate selection can be based only on the required terms, which also have shorter posting lists. Once a candidate is selected, the terms are evaluated in order and the optional term iterators are advanced with a skip. Finally, at any point, a partially scored candidate can be pruned if its partial score plus the maximum remaining contribution is below the score of the current $k$-th best candidate, i.e., $s + a_i < \tilde{s}$. The description is so far similar to [5].

Now we explain how to apply these ideas to pipelined query processing. At query processing time, the broker fetches maximum scores along with term ID and location information and includes them in the query bundle. Therefore, when $b_q$ arrives at a particular node, the information about the maximum scores of the terms in the current sub-query, $\hat{s}_t$ s.t. $t \in q_j$, and the maximum contribution of the remaining sub-queries, $\tilde{a} = \sum \hat{s}_t$ s.t. $t \in q_i$, $q_i \subseteq q$ and $i > j$, are available to the query processor. $b_q$ itself is routed by decreasing maximum $\hat{s}_t$ among the sub-queries. Each query processor treats the received accumulator set just as a posting list iterator with $\hat{s}_t$ set to highest score within the set, and processes the sub-query document-at-a-time. Therefore, the required subset is defined by $a_r \geq \tilde{s} - \tilde{a}$, any candidate

**Algorithm C.2**: processBundle($b_q$) with DAAT Max-Score optimization

**Data**: iterators $\{i_1, \ldots, i_l\}$ and maximum scores $\{\hat{s}_1, \ldots, \hat{s}_l\}$ sorted by descending $\hat{s}$, accumulator set $A$, bundle attributes $\tilde{s}$ and $\tilde{a}$

1 **if** $A \neq \emptyset$ **then** $l \leftarrow l+1$, **for** $x \leftarrow l$ *to* $1$ **do** $i_x \leftarrow i_{x-1}$, $\hat{s}_x \leftarrow \hat{s}_{x-1}$ **endfor**, set $i_1$ to be $A$'s iterator and $\hat{s}_1$ to be $A$'s maximum score;
2 $a_l \leftarrow \hat{s}_l$, **for** $x \leftarrow l-1$ *to* $1$ **do** $a_x \leftarrow a_{x+1} + s_x$ **endfor**, $A' \leftarrow \emptyset$, *minHeap* $\leftarrow \emptyset$;
3 $r \leftarrow l$, **while** $r > 0$ *and* $a_r < \tilde{s} - \tilde{a}$ **do** $r \leftarrow r-1$;
4 $d' \leftarrow -1$;
5 **while** $r > 0$ **do**
6 advance iterators having $i_{x \leq r}.d() = d'$, if $i_x.next() =$ *false*: close $i_x$, update $i$, $s$ and $a$ sets, decrement $l$, recompute $r$ (similar to line 3, break if $r = 0$);
7 $d' \leftarrow \min(i_{x \leq r}.d())$, $s \leftarrow 0$;
8 **for** $x \leftarrow 1$ *to* $l$ **do**
9 **if** $s + a_x < \tilde{s} - \tilde{a}$ **then** break and proceed to selection of the next candidate (line 5);
10 **if** $x > r$ *and* $i_x.d() < d'$ **then** advance $i_x$ to $d'$, if $i_x.skipTo(d') =$ *false*: close $i_x$, update $i$, $s$, $a$, $l$ and $r$ (break if $r = 0$) and proceed to the next iterator (line 8);
11 **if** $i_x.d() = d'$ **then** $s \leftarrow s + i_x.\mathrm{s}()$;
12 **if** $s \geq \tilde{s} - \tilde{a}$ **then** add $\langle d', s \rangle$ to $A'$;
13 **if** $s >$ *minHeap.minScore* **then** add $\langle d', s \rangle$ to *minHeap*, $\tilde{s} \leftarrow \max(\tilde{s}, s)$;
14 **if** it is the last node in the route **then**
15 retrieve candidates from *minHeap*, sort and return them to the broker;
16 **else** $A \leftarrow \{\langle d, s \rangle \in A' \text{ s.t. } s \geq \tilde{s} - \tilde{a}\}$, update $b_q$ with $A$ and $\tilde{s}$ and send it to the next node;

can be pruned whenever $s + a_i < \tilde{s} - \tilde{a}$ holds, and finally, the candidates with partial scores $s \geq \tilde{s} - \tilde{a}$ have to be transferred to the next node as a modified accumulator set.

We use Algorithm C.2 to describe the final query processing approach. First, the algorithm prepares for query processing, calculates $a$ values and defines the required set (lines 1-4). As long as the required set is non-empty, the iterators are processed document-at-a-time (lines 5-13). Each iteration advances the recently used iterators of the required set, selects a new candidate (lines 6-7) and accumulates its score (lines 8-11). If an iterator reaches the end of the posting list, it is removed from the iterator set and the $a$ values of remaining iterators and $r$ are updated (lines 6 and 10). If a candidate succeeds to reach a score higher than the pruning threshold ($s \geq \tilde{s} - \tilde{a}$), it is inserted in the accumulator set (line 12). Potentially, it may also be inserted into the candidate heap (line 13). In this case, $\tilde{s}$ may also be updated. When a sub-query is fully processed, if this is the last node in the route, the candidate heap has to be sorted and returned to the broker as the final result set (lines 14-15). Otherwise, non-pruned accumulators have to be transferred to the next node. In this case, prior to the final transfer, an extra pass through the accumulator set removes candidates having $s < \tilde{s} - \tilde{a}$, which are false positives due to a monotonically increasing pruning threshold. In order to facilitate pruning, $\tilde{s}$ is initiated with the value received from the previous node, or $0$ for the first node. In practice, the last node in the route does not have to store non-pruned accumulators, but only the candidate heap.

## C.6 Max-Score optimization of term assignment

The pruning performance of the Max-Score optimization can be limited when long posting lists appear early in the pipeline. Therefore, we suggest to assign posting lists by decreasing $\hat{s}_t$, such that the first node gets posting lists with highest $\hat{s}_t$ and the last node gets posting lists with lowest $\hat{s}_t$. For simplicity, we use equally sized partitions. Since $\hat{s}_t$ increases as $f_t$ decreases (because most of the similarity functions, including BM25, use the inverse document frequency), this strategy implies that the first node now stores only short posting lists and the last node stores a mix of long and short posting lists, and the nodes in between store posting lists with short-to-moderate lengths.

As we show in the next section, this technique significantly improves the performance, but struggles with a high load imbalance. Beyond the experiments presented in the next section, we have tried several load balancing approaches similar to Moffat et al. [9], such as estimating the workload associated with each term $t$ in a past query log $Q'$, $L_t = |I_t| \times f_{t,Q'}$, where $f_{t,Q'}$ is the frequency of $t$ in $Q'$ and $|I_t|$ is the size of $I_t$, and splitting the index so the accumulated past load would be balanced across the nodes. However, since the load estimator does not take skipping into account, it overestimates the load of long posting lists and assigns nearly half of the index to the first node. As a result, the load imbalance gets only worse. Therefore, we leave load balancing as an important direction for further work and outline a possible solution in Section C.8.

## C.7 Experimental results

For our experiments, we index the 426GB TREC GOV2 corpus. With stemming and stopword removal applied, it contains 15.4 mil. unique terms, 25.2 mil. documents, 4.7 bil. pointers and 16.3 bil. tokens. With 8 index partitions, the resulting distributed index is 9.3GB in total, while a corresponding monolithic index is 7.6GB. Most of the overhead (1.54GB) comes from a short replicated version of the document dictionary. Skipping pointers increase the size by additional 87.1MB and the resulting index contains 279 647 posting lists with one skip level, 15 201 with two and 377 with three levels.

We run our experiments on a 9 node cluster. Each node has two 2.0GHz Quad-Core CPUs, 8GB memory and a SATA disk. The nodes are interconnected with a Gigabit network. Our framework[2] is implemented in Java. It uses Java NIO and Netty for efficient disk access and network transfer. For disk access, we use 16KB buffers and the default GNU/Linux OS caching policy (hence, we reset the disk caches before each run). Further, queries are preprocessed in the same way as the document collection and evaluated using the Okapi BM25 model.

We evaluate the following query processing methods. Full/non-pruned evaluation (Full), space-limited pruning described in Section C.4 (LT denotes the state-of-the-art method, and SLT denotes the skip-optimized version), Max-Score optimized evaluation described

[2] `https://github.com/s-j/laika`

Table C.1: Impact of the query processing method on the precision and recall of the Adhoc Retrieaval Topics 701-850.

| Method | MAP | P@10 | Recall |
|---|---|---|---|
| Full/MSD | .153903 | .530872 | .274597 |
| LT1M | .153746 | .530872 | .274262 |
| LT100K | .152376 | .528859 | .270955$^{\dagger}$ |
| LT50K | .152378 | .530872 | .271049 |
| LT25K | .150602$^{\dagger}$ | .528188 | .267361$^{\dagger}$ |
| LT10K | .145877$^{\ddagger}$ | .516107$^{\dagger}$ | .256386$^{\#}$ |
| AND | .155073 | .531544 | .268126 |

Table C.2: Maximum and average document frequency and sample covariance between the document and query frequency distributions in the evaluated query sets.

| Query set | $\max(f_t)$ | $\text{avg}(f_t)$ | $\text{cov}(f_t, f_{t,Q})$ |
|---|---|---|---|
| A04-06 | 11256870 | 997968 | 149393 |
| E05 | 11256870 | 223091 | 2266801 |
| E06 | 11256870 | 290747 | 5999742 |

in Section C.5 (MSD), and finally an evaluation with intersection semantics and document-at-a-time sub-query processing (AND). We use a subscript N (e.g., $\text{SLT}_\text{N}$) to denote an execution on a non-optimized index and S on a self-skipping index. To limit the number of experiments, the maximum number of top-results $k$ is fixed at 100. For LT we vary the accumulator set target size $L$, hence LT1M corresponds to $L = 1\ 000\ 000$ and LT10K to $L = 10\ 000$. Finally, we fix the $h_{\max}$ used by LT/SLT (see Sec. C.4) at 2000, which we find to be suitable for our index.

In order to evaluate the impact of the query processing optimizations on the retrieval performance, we use the TREC Terabyte Track Adhoc Retrieval Topics and Relevance Judgments 701-850 from 2004, 2005 and 2006. We use documents with relevance judgments 1 and 2 as a ground truth and consider MAP, precision at 10 results (P@10) and recall at $k$ results as retrieval performance indicators. Table C.1 shows the averages over the whole query set. We focus on result degradation with the space-limited pruning (LT) compared to a full evaluation (Full), which is the retrieval performance baseline. Beyond the average results, we apply a paired t-test at the query level to check the degree of significance. We use † to mark the significance at 0.05 level, ‡ at 0.01 and ♯ at 0.001.

Table C.1 shows how the retrieval performance of LT degrades with decreasing $L$. Degradation becomes significant at lower values of $L$. The evaluation measures of LT50K and LT100K are different, but without statistical significance when compared to each other. Beyond the presented data, the results obtained with Full and LT10M are identical, and the results obtained with SLT are identical to LT for $L \geq 10\ 000$. From these results, we consider $L \geq 50\ 000$ as a suitable choice with respect to the precision and recall with $k = 100$ (while $L \leq 25\ 000$ is not) and keep LT100K, LT50K and LT25K for further experiments.

Our observations confirm that the results obtained by Full and MSD are identical. Furthermore, we observe a higher precision (MAP and P@10) with AND compared to Full,

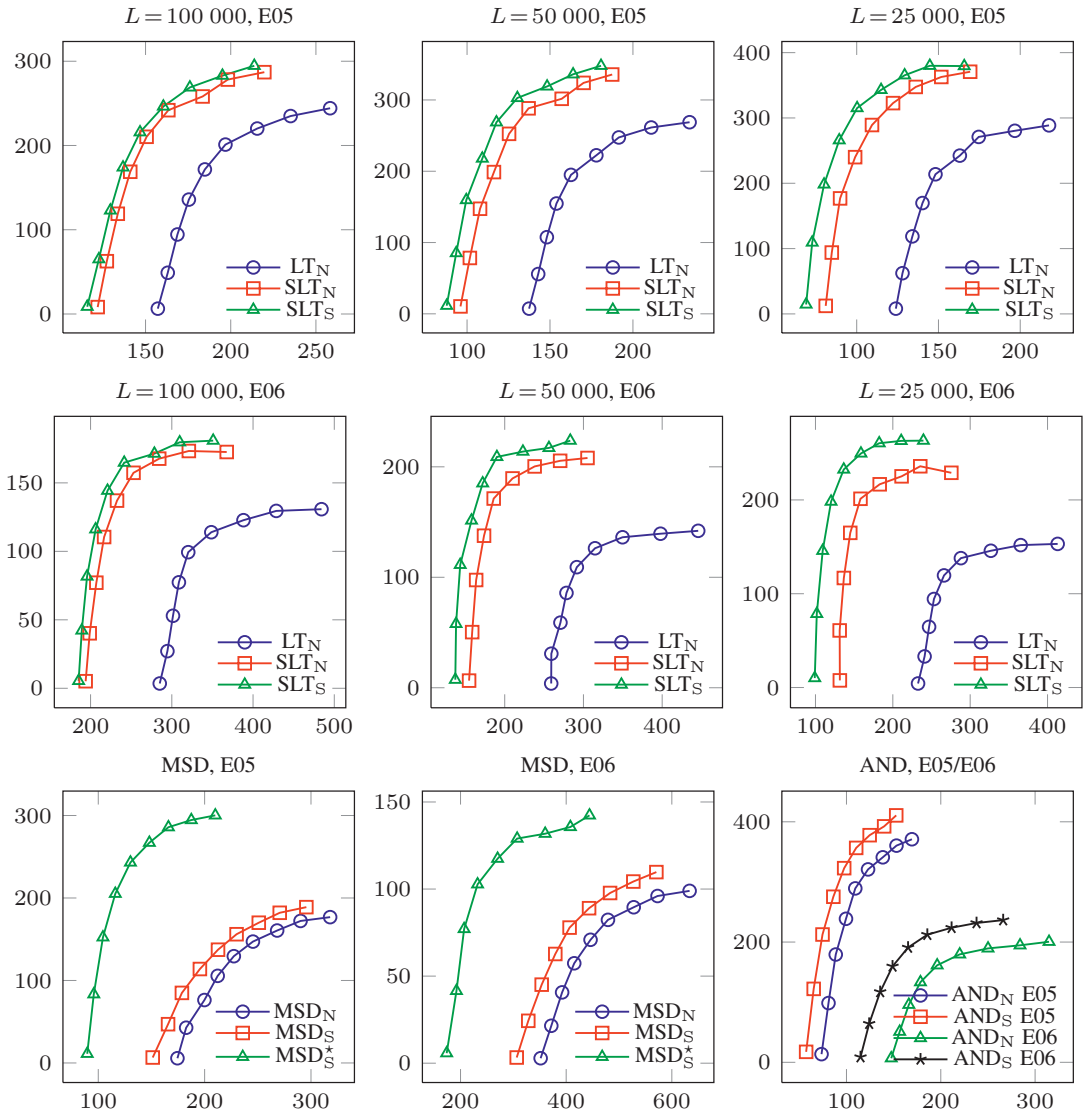



Figure C.1: Throughput (y-axis, qps) and latency (ms) with varied multiprogramming.

although the difference is statistically insignificant. A closer look has shown that AND performs better for topics 701-751 and 751-800 and worse for topics 801-850 (no significance). However, with $k = 1000$, Full has both a higher MAP (0.271470 versus 0.255441, no significance) and a higher recall (0.662522 versus 0.596165, significance at 0.01).

In order to evaluate the algorithmic performance, we use two subsets of TREC Terabyte Track Efficiency Topics from 2005 (E05) and 2006 (E06). Both subsets contain 20 000 queries that match at least one indexed term, where the first 5 000 queries are used for warm-up and the next 15 000 to measure the actual performance. To simulate the effect of a result cache, neither of the sets contains duplicated queries.

We observe that E06, which contains government-specific queries, implies higher query processing load than E05, which contains Web-specific queries. Therefore, we consider these two sets as a better (E05) and a worse case (E06) scenarios. We use Table C.2

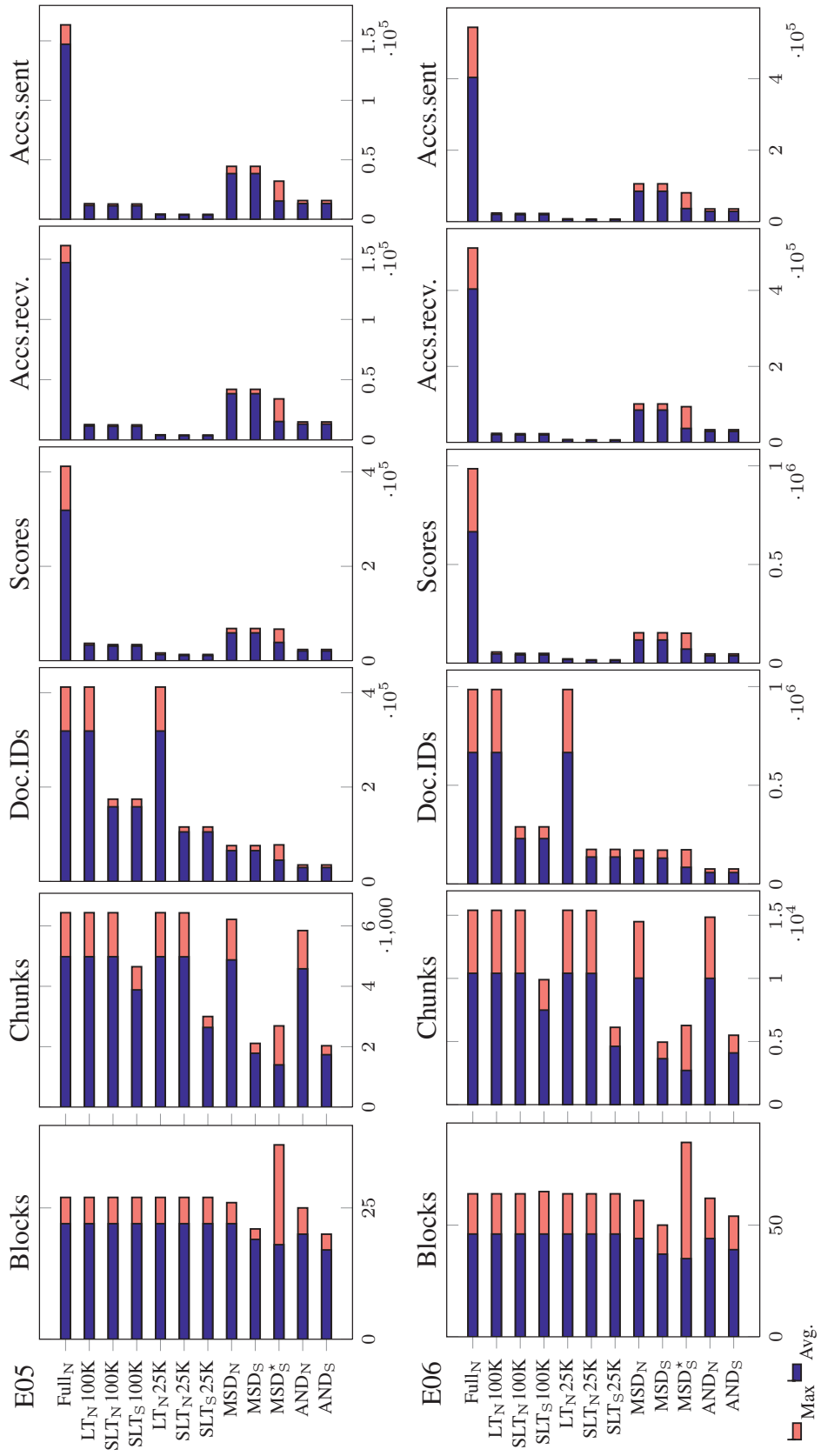



Figure C.2: Maximum and average number of processed entities per node.

to illustrate the difference between the query sets, including the Adhoc topics marked as A04-06. As the table shows, the term with highest document frequency (i.e., posting list length) is the same in all three of the sets (which is the term 'state'), but the average document frequency and the sample covariance between the document frequency and the query set frequency are significantly different. A04-06 is a very short query set, with a flat query frequency distribution and missing a long-tail distribution among the document frequencies. This explains a high average document frequency and a low covariance. Finally, the results for E05 and E06 illustrate the load difference between these two sets. E06 contains terms with both longer posting lists and a higher correlation between the query and document frequencies.

Figure C.1 illustrates the average latency per query (milliseconds) and overall throughput (queries per second) with varied multiprogramming levels. Points on each plot correspond to 1, 8, 16, 24, 32, 48, 56 and 64 concurrent queries (cq). Each run was repeated twice (with the disk cache being reset each time) and the average value was reported. We report results for all methods except Full, which was too slow – even when the multiprogramming level set to 1, a query took on average 474ms for E05, and 1121ms for E06. Therefore, we consider $\mathrm{LT_N}$ as the time performance baseline. Our results show that for both E05 and E06, the skipping optimization to LT ($\mathrm{SLT_N}$) significantly improves the performance and the improvement increases with smaller $L$. Skipping support in the inverted index ($\mathrm{SLT_S}$) provides a further improvement. While the index optimization is not as significant as the algorithmic optimization, we observe that (for E06) the improvement increases as $L$ decreases. We explain these results using Figure C.2, which illustrates the number of blocks read, chunks decompressed, unique document IDs evaluated, scores computed and accumulators sent and received by each node, normalized by the evaluation set query count. The figure shows both the average (across the nodes) and the maximum (one node) counts. As the results show, the improvement from $\mathrm{Full_N}$ to $\mathrm{LT_N}$ lies in reduced score computation and network transfer, which improve as $L$ decreases. $\mathrm{SLT_N}$ further improves the number of candidates been considered (Doc.IDs) and $\mathrm{SLT_S}$ improves the amount of data been decompressed (Chunks). However, even with $L = 25\ 000$ there is no additional savings in data read from disk (Blocks), which can be explained by a relatively large block size (16KB).

As we show in Figure C.1, the Max-Score optimization ($\mathrm{MSD_N}$) gains a modest improvement from the self-skipping index ($\mathrm{MSD_S}$) and a significant improvement from the further term-assignment optimization ($\mathrm{MSD^\star_N}$). For E05, $\mathrm{MSD^\star_N}$ performs as good as $\mathrm{SLT_S}$100K, but for E06 it struggles with increasing query latency when compared to $\mathrm{SLT_S}$100K. However, having in mind that MSD is equivalent to a full (non-pruned) evaluation, it shows a very good performance. As Figure C.2 shows, the Max-Score optimizations significantly improve the total amount of read, decompressed and evaluated data. While these methods increase the number of score computations and the number of transferred accumulators compared to the LT optimizations, they show a significant improvement compared to Full. Finally, our results show that the main challenge of $\mathrm{MSD^\star_N}$ is an increased load imbalance, which is the ratio between the maximum and the average counts. This issue should be investigated in future.

As illustrated in Figures C.1 and C.2, skipping support in the index improves the perfor-

mance of AND by reducing the amount of read, decompressed and evaluated data. For E06 it improves the latency at 1cq by 22% and the throughput at 64cq by 18%, for E05 it improves the latency at 1cq by 29% and the throughput at 64cq by 11%. Overall, AND performs better than $\text{SLT}_\text{S}$25K on E05 and slightly worse on E06. As having a better result quality than LT25K on A701-850, we consider AND as a good alternative to the space-limited pruning with a low accumulator set target size ($L$).

Finally, we address performance linearity in Figure C.3. In these results, we keep the multiprogramming level at 64cq and vary the collection size to $1/2$ and $1/4$. For LT/SLT we use $L = 100\ 000$ scaled with the collection size. The results show that with increasing index size, the methods converge in the absolute throughput and diverge in the latency. In our opinion, the best behavior is given by SLT, MSD and AND. However, our results do not guarantee the performance for a collection larger than the GOV2. This should be addressed in future.

## C.8 Conclusions and further work

We have presented and evaluated several skipping optimizations to pipelined query processing. For $\text{SLT}_\text{N}$ and $\text{SLT}_\text{S}$ our results indicate a significant improvement over the baseline approach. We also came up with a pruning approach ($\text{MSD}^\star_\text{S}$) that provides a result quality equivalent to a non-pruned evaluation, while having a considerably good performance. Further, we have observed that processing queries with conjunctive semantics (AND) provides good retrieval performance and efficient query processing. Although the state-of-the-art approach considers disjunctive (OR) queries, in future, we would like to take a closer look at AND queries. Finally, based on our current results, we outline three techniques that can further improve the performance of $\text{MSD}^\star_\text{S}$:

**Dynamic load balancing.** The load balancing of $\text{MSD}^\star_\text{S}$ can be improved by gradually moving the posting lists with the highest or the lowest $\hat{s}$ values to one of the neighbouring nodes. Compared to the previously presented fill-smallest and graph-partitioning techniques, this approach will reduce the network volume at repartitioning and can be done dynamically. In order to avoid moving data back-and-forth, we can further replicate the bordering posting lists and fine-tune partitions at the lexicon level, without actual repartitioning.

**Hybrid query processing.** $\text{MSD}^\star_\text{S}$ tends to place the shortest posting lists (corresponding to rare terms) on the first nodes. Therefore, by transferring these (complete) posting lists to a node corresponding to a later sub-query we can remove decompression, processing and accumulator transfer from the first few nodes (with an additional opportunity for parallelism). The node receiving the posting lists will in this case substitute an accumulator set with a few short posting list. In order to minimize the load on the sending node and the overall network load, the nodes can further cache the received lists.

**Intra-query concurrent processing.** Document-at-a-time processing of sub-queries allows to transfer accumulators to the next node as soon as possible. This feature can be utilized to provide intra-query concurrency and improve the performance at low multiprogramming levels. In [6], we have already evaluated an extension to an earlier version

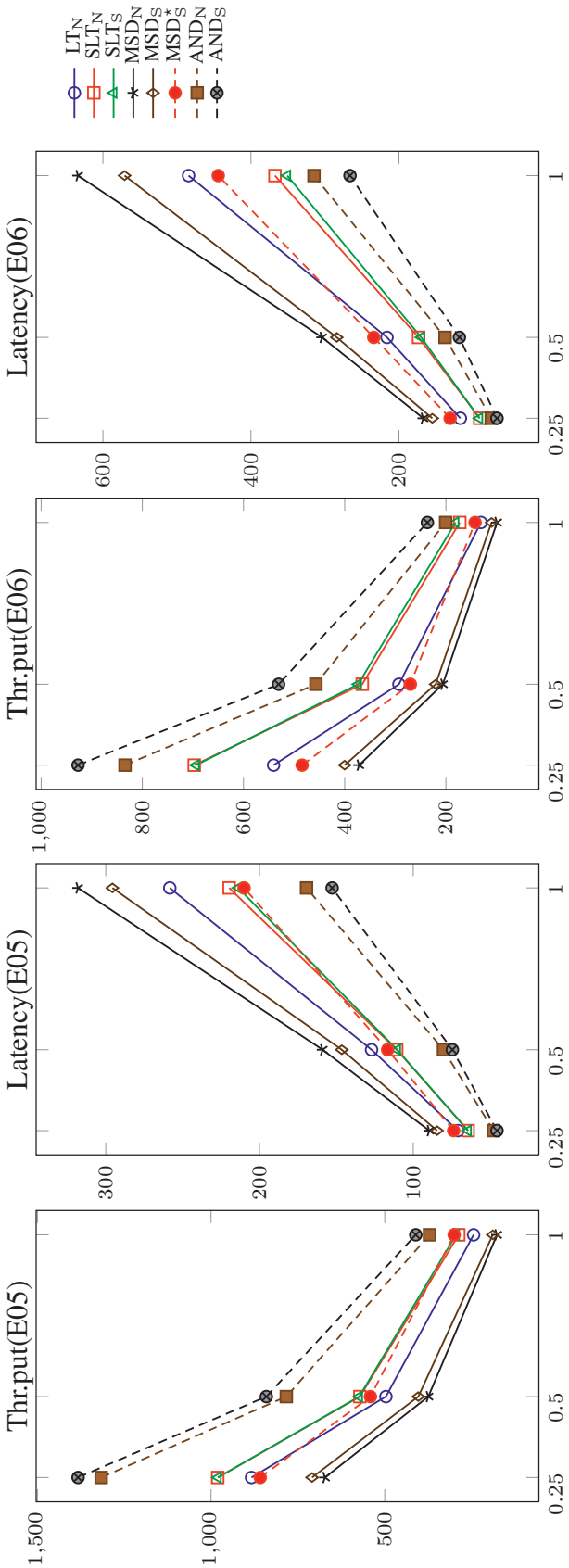



Figure C.3: Throughput (qps) and latency (ms) with varied collection size (x-axis).

of $\text{MSD}^{\star}_{\text{S}}$. The optimization splits the document ID range into several sub-ranges, called fragments, and does intra-query parallelization at the fragment level, both across the nodes and on the same node. The experiments [6] with a smaller subset of the TREC 2005 Efficiency Topics indicated that this optimization allows to reach a similar peak-throughput at nearly half of the latency. We assume this to be applicable to our current method, however a further evaluation for the 2006 topics is needed.

**Acknowledgments.** This work was supported by the iAd Centre and funded by the Norwegian University of Science and Technology and the Research Council of Norway.

## C.9 References

[1] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. 2010.

[2] B. B. Cambazoglu and C. Aykanat. A term-based inverted index organization for communication-efficient parallel query processing. In *IFIP NPC*, 2006.

[3] S. Jonassen and S. E. Bratsberg. Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes. In *NIK*, 2009.

[4] S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *WISE*, 2010.

[5] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.

[6] S. Jonassen and S. E. Bratsberg. Intra-query concurrent pipelined processing for distributed full-text retrieval. In *ECIR*, 2012.

[7] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 2005.

[8] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale*, 2007.

[9] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, 2006.

[10] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 2007.

[11] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, 2005.

[12] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Proc. and Manag.*, 1995.

[13] W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, University of Melbourne, 2007.

[14] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.

[15] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *IPDPS*, 2007.

Appendix D

# Paper A.IV: Intra-Query Concurrent Pipelined Processing For Distributed Full-Text Retrieval

Simon Jonassen and Svein Erik Bratsberg.
*Appeared at the 34th European Conference on Information Retrieval (ECIR), Barcelona, Spain, April 2012.*

**Abstract:** Pipelined query processing over a term-wise distributed inverted index has superior throughput at high query multiprogramming levels. However, due to long query latencies this approach is inefficient at lower levels. In this paper we explore two types of intra-query parallelism within the pipelined approach, parallel execution of a query on different nodes and concurrent execution on the same node. According to the experimental results, our approach reaches the throughput of the state-of-the-art method at about half of the latency. On the single query case the observed latency improvement is up to 2.6 times.

## D.1 Introduction

With a rapid growth of document collections and availability of cheap commodity workstations, distributed indexing and query processing became the most important approach to large-scale, high-performance information retrieval. Two underlying, fundamentally different methods are term-wise and document-wise partitioning. With a large number of controversial comparisons and several hybrid methods presented throughout the last 20 years, both methods have their advantages and challenges. Term-wise partitioning, which we address in this paper, reduces the number of disk seeks [13] and improves inter-query concurrency [1].

The traditional query processing approach to a term-wise distributed index is to use one of the nodes as a ranker node and the remaining nodes as fetchers. With document-ordered inverted files this leads to a high network load and a large ranker overhead. In order to overcome these problems, pipelined query processing [12] suggests to create a query bundle, which includes the query itself and a set of partially scored documents, accumulators, and route it through the nodes hosting the posting lists associated with the query. At each node the accumulator structure is modified with more posting data, and at the last node the $k$ top-scored accumulators are selected, sorted and returned as a final result. Fig. D.1(a) illustrates the execution of a single query.

With several optimizations [11, 17] to load balancing and replication of the most load consuming posting lists, pipelined approach has been shown to outperform document-wise partitioning in terms of query throughput. However, this comes at a cost of long query latency. Fig. D.2 reconstructs the results presented by Webber [17], one of the authors of pipelined approach, who provided a detailed description of the methods and experiments done in their work. The figure shows that the method has a high maximum throughput, but it is less efficient at lower query multiprogramming levels (i.e. when the number of queries processed concurrently is small). For a user-oriented search application it is important to keep the latency low and be efficient at both high and low query loads. In this paper we look at the intra-query parallelism possible with a modification to pipelined approach. Our objective is to reduce query latency at lower multiprogramming levels, while keeping the performance degradation at higher levels minimal. As the main goal we want to achieve the same throughput at a significantly lower latency.

Our contribution is as follows. We address the intra-query concurrency problem of pipelined query processing. We present a novel technique that exploits intra-query parallelism between and within each node. We evaluate our experiments on a real distributed system, using a relatively large document collection and a real query set. Finally, we suggest several directions for the future work.

## D.2 Related work

Performance comparison studies of the document- and term-wise partitioning methods have been presented in a large number of publications. In this paper we refer only to

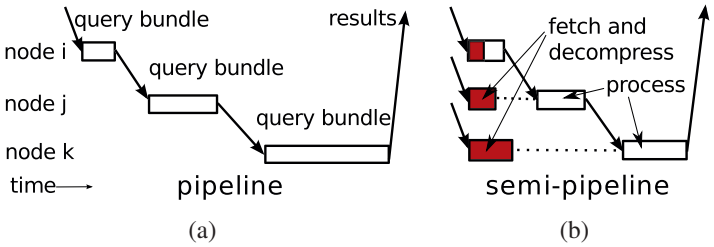



Figure D.1: (a) Pipelined and (b) semi-pipelined query processing.

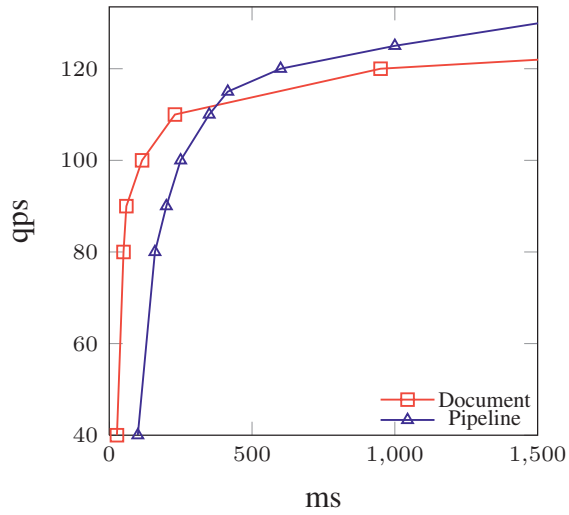



Figure D.2: Reconstruction of the latency/throughput with varied multiprogramming levels presented by Webber [17].

a few of them [1, 4, 9, 11, 12, 13, 18]. Several works [4, 18] have also presented and evaluated hybrid partitioning and query processing strategies. Early optimizations of distributed query processing considered conjunctive (AND) queries and using the shortest term to eliminate unnecessary postings [15]. Several other studies, such as the work done by Marin *et al.* [9, 10], have used impact- or frequency-ordered lists. In our work, we look at disjunctive (OR) queries and document-ordered indexes. While impact-ordered indexes offer highly-efficient query processing [14], document-ordered indexes combined with careful document ID ordering [19], skipping and MaxScore [6, 16] or WAND [3] style processing are highly-efficient as well. At the same time, document-ordered indexes are easier to maintain and process.

Pipelined query processing (P) was presented by Moffat *et al.* [11, 12] and Webber [17]. According to the original paper [12] this method significantly improves throughput, but struggles with load imbalance and high network load. Several query-log-based term-assignment methods have been suggested in order to improve load-balancing [11, 17], reduce communication cost [20], or both [8]. The results presented by Webber [17] show that, due to inter-query parallelism, P succeeds to achieve higher throughput than document-wise partitioning once the load balancing issues are resolved. However, the

author admits lacking intra-query parallelism, resulting in a poor performance under light-to-moderate workloads, and suggested that preloading of inverted lists would enable disk parallelism and therefore improve the performance.

According to Büttcher *et al.* [2], two other problems of P lie in Term-at-a-Time (TAAT) processing and a poor scalability with collection growth. The original implementation of P uses the space-limited pruning method by Lester *et al.* [7], which allows to restrict the number of transferred accumulators (thus reducing the network and processing load), but requires a complete, TAAT processing of posting data.

In our recent work [5] we have presented a combination of parallel posting list prefetching and decompression and pipelined query processing, called semi-pipelined query processing. We illustrate this approach in Fig. D.1(b). Additionally, our previous work included execution of some of the queries in a traditional, non-pipelined way, and a different query-routing strategy. The results reported 32% latency improvement, but the underlying model assumed that each posting list is read and decompressed *completely* and *at once*.

As the baseline for the current work we use a modification of P applying inverted index skipping, MaxScore pruning and Document-at-a-Time (DAAT) processing within each sub-query, which we briefly describe in the next section. The intention behind this is to tackle the issues addressed by Büttcher *et al.*. The underlying query processing on each node and the index structure itself are similar to those presented in the recent work on inverted index skipping [6]. While our baseline processes different posting lists in parallel (DAAT), the bundle itself is processed by one node at a time, which is the main reason for long query latencies and a poor performance at the low query multiprogramming levels. As inverted index skipping makes semi-pipelined processing impossible, we suggest that intra-query parallelism between different nodes and within each posting list is the best way to improve the query processing performance. Finally, as we substitute space-limited pruning [7] with MaxScore, the quality of query results is equivalent to a full, disjunctive query evaluation.

## D.3 Preliminaries

For a given document collection $D$ and a query $q$, we look at the problem of finding the $k$ top-ranked documents according to a similarity score $sim(d, q) = \sum_{t \in q} sim(f_{d,t}, D, q)$. Here, $sim(f_{d,t}, D, q)$ or simply $s_{d,t}$ is a term-similarity function, such as Okapi BM-25 or TF$\times$IDF, and $f_{d,t}$ is the number of occurrences of the term $t$ in the document $d$.

**Skipping.** For any term $t$ in the inverted index, the posting list $I_t$ contains a sequence of document IDs and corresponding frequencies. Within each list postings are ordered by document ID, divided into groups of 128 entries (chunks) and compressed with NewPFor [19] using gap-coded document IDs. A hierarchy of skipping pointers, which are also gap-coded and compressed in chunks, is built on top. The logical tree is then written to disk as a prefix-traverse. A posting list iterator accessing the resulting index reads the data block-wise and keeps one chunk from each level (decompressed) in the main memory. The combination of bulk-compression, reuse of the decompressed data, index layout and buffering results in highly efficient query processing.

**Distributed index.** In order to create a distributed, term-wise partitioned index we sort posting lists by their decreasing maximum scores $\hat{s}_t = \max_{d \in I_t}(s_{t,d})$. Then we assign them to $n$ different worker nodes in a such way that the node $i$ receives the posting lists with $\hat{s}_t$ higher than those received by the node $i+1$, but lower than $i-1$, and the partitions have nearly the same size. Additional data structures such as a small local lexicon and a replica of a short document dictionary are stored on each node. An additional node $n+1$, which serves as a query broker, stores a full document dictionary and a global lexicon. During the query processing, the only structure accessed from disk is the inverted index, all the other structures are kept in the main memory as sorted arrays and accessed by binary search.

**Query processing.** At query time, each query is received, tokenized, stop-word processed and stemmed by the query broker. The resulting terms are checked in the global lexicon and the collection-based $\hat{s}_t$ values are adjusted with the normalized number of occurrences in the query. Further, the query is divided into a number of sub-queries, each containing only the terms assigned to the particular node, and a route is chosen by decreasing maximum $\hat{s}_t$ in each sub-query. Finally, the broker generates a bundle message and sends it to the first node in the route.

The bundle is processed by one node at a time. Each node in the route receives the bundle, decompresses the received accumulator set, matches it against its own posting data, generates a new accumulator set, compresses and transfers it to the next node. Query processing on each node is similar to the traditional DAAT MaxScore [16], except that it is limited only to the received accumulator set and the query-related posting lists stored on this node. Therefore, it operates with a pruning score threshold $v = \mathrm{minHeap.min} - r$. Here, $r$ is the accumulated maximum score of the terms in the remaining sub-queries and minHeap.min is the smallest score within the $k$ top-scored results seen so far (monitored with a heap). Any partially scored accumulator can be pruned at any time if its estimated full score falls below the current value of $v$. Additionally, some of the posting lists cannot create new accumulators and therefore can be processed in a skip-mode. Accumulators that cannot be pruned have to be transferred to the next node. As $v$ increases during processing, more posting data can be skipped and more existing accumulators can be eliminated. The last node in the route does not have to create a new accumulator set. Instead, it uses only the candidate heap and when processing is done, it extracts, sorts and returns the final candidates to the broker as the result set.

Accumulators that pass the threshold are placed into a new accumulator set. Since $v$ increases within each sub-query, the accumulators in the beginning of the set may have scores below the final value of $v$, $v_{\mathrm{final}}$. We call these false positives. In order to eliminate them, when $v_{\mathrm{start}} < v_{\mathrm{final}}$, an additional pass through the accumulator set is done in order to preserve only those having scores $s \geq v_{\mathrm{final}}$. $v_{\mathrm{final}}$ is transferred along with the query bundle and used as $v_{\mathrm{start}}$ on the next node in order to facilitate pruning. Next, $v$ is updated with a new value only when a new accumulator has been inserted into the candidate heap and $v < \mathrm{minHeap.min} + r$. Finally, prior to a transfer the accumulator IDs are gap-coded and compressed with NewPFor and the partial scores are converted from double to single precision.

## D.4 Intra-query parallel processing

In this section we introduce our new query processing approach, divided in two parts. In the first part we address intra-query parallelism on different nodes, and in the second part - on the same node. The experimental results and comparison to the baseline approach follow in the next section.

### D.4.1 Query parallelism on different nodes

**Fragment pipeline.** In order to overlap the execution of the same query on two consecutive nodes, we divide the document ID range into several sub-ranges, fragments. For a query $q$ we define a fragment size $F_q$, which splits the ID range $[0, |D|)$ into $N_q = \lceil \frac{|D|}{F_q} \rceil$ sub-ranges or fragments. Fragment $i$ covers document IDs $[iF_q, (i+1)F_q)$.

As we illustrate in Fig. D.3(a), each sub-query can now be divided into several tasks, each processing the sub-query over a single fragment. For simplicity, we explain the execution of a single query. With the state-of-the-art approach each sub-query is processed as a single task, which includes three steps: (a) decompression of the incoming accumulator set, (b) processing/merging of the posting and accumulator data, and (c) elimination of false-positives and compression of the new accumulator set or extraction of final results. With a fragment-pipeline, all three steps are scaled down to a single fragment. For example, the node processing the first sub-query post-processes, compresses and transfers its partial accumulator set as soon as the first fragment is finished. Then it starts straight on the second fragment. The next node in the route starts processing as soon as the accumulator set corresponding to the first fragment has arrived.

As an alternative to this method we could process each sub-query until the number of non-pruned accumulators would be above a minimum number, then transfer these and resume processing. However, this could lead to one-to-many and many-to-one correspondences between processing tasks on different nodes, and therefore require a complex implementation with many special cases. Our solution simplifies the implementation, as each node has only one-to-one correspondence between incoming and outgoing fragments. Additionally, we avoid delaying the accumulator transfer in order to wait for the next incoming fragment.

So far we look at the processing model where all tasks corresponding to a single sub-query are done by a single thread, the executor. In order to be efficient, these tasks have to reuse the candidate heap, the pruning threshold and the state of the posting list iterators, called sub-query state. The state contains information on the number of non-finished iterators, including the current position within the posting list, recently decompressed and fetched data (which can be reused by future tasks), and which of the posting lists can be processed in the skip-mode. Further, as posting list iterators support only next() and skipTo($d$) operations, fragments have to be processed in-order. As Fig. D.3(c) shows, a priority queue and a counter are associated with each sub-query to enforce the order. As fragments arrive, they are inserted into the priority queue and the corresponding executor is notified. If the next fragment in the priority queue has the sequence ID corresponding to the counter

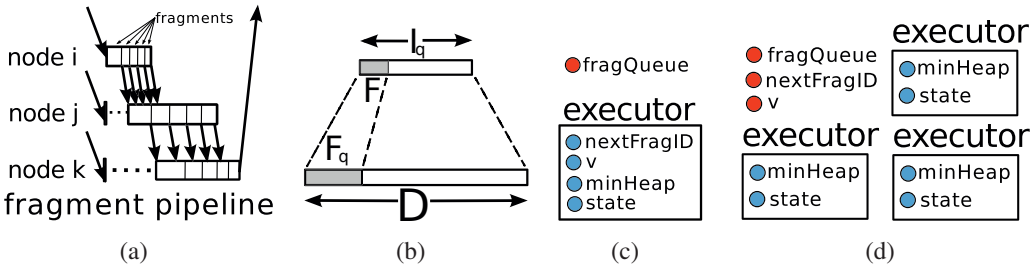



Figure D.3: (a) Fragment pipelined query processing. (b) Mapping between $F$ and $F_q$. (c)-(d) Data structures used by (c) non-concurrent and (d) concurrent fragment processing.

value, the fragment is processed by the executor and forwarded to the next node, and the counter is increased. If not, the executor suspends processing in order to wait for more data.

Additionally, as the pruning threshold for each sub-query increases gradually, at some point the current pruning threshold of a sub-query $i$, $v_i$, can exceed the current threshold value in the next sub-query $i+1$. Therefore, $v_i'$ seen right after finishing a fragment is packed and transferred along with the accumulators. On the next node, it replaces the current threshold $v_{i+1}$ if $v_i' > v_{i+1}$.

**Fragment size estimation.** As different queries have different processing cost, those more expensive queries are desired to consist of a larger number of fragments than the shorter ones. We assume that the total processing cost of each query is proportional to the total number of candidates produced by a full non-pruned disjunction of query terms, $I_q = \bigcup_{t \in q} I_t$. Next, we introduce a system-dependent (smallest) fragment size $F$, which corresponds to the fragment size used by some hypothetical query that has to consider all of the documents in the collection as potential accumulators. $F$ has to be chosen dependent on the systems settings. In practice it can be tuned during the warm-up or in the run-time. For a particular query $q$, the fragment size $F_q$ can be chosen so that $|I_q|/F = |D|/F_q$ holds. As Fig. D.3(b) shows, this equality reflects the correspondence between the document ID space $D$ and the results set $I_q$.

Assuming non-correlated terms, $|I_q|$ can be approximated by Eq. (D.1). The equation uses the probability that a document does not contain a given term $t$, $(1 - \frac{|I_t|}{|D|})$, to find the probability that a document contains at least one of the query terms, and finally multiplies it by the total number of indexed documents $D$. Then, $F_q$ can be calculated with Eq. (D.2). As $F_q$ cannot be smaller than $F$ or larger than $|D|$, we further apply Eq. (D.3).

$$|I_q| \approx |D| \cdot (1 - \prod_{t \in q} (1 - \frac{|I_t|}{|D|})) \tag{D.1}$$

$$F_q' = \frac{|D|}{|I_q|} \cdot F \approx F/(1 - \prod_{t \in q} (1 - \frac{|I_t|}{|D|})) \tag{D.2}$$

$$F_q = \begin{cases} F & \text{if } F_q' < F \\ |D| & \text{if } F_q' > |D| \\ F_q' & \text{otherwise} \end{cases} \tag{D.3}$$

### D.4.2 Sub-query parallelism on a single node

**Concurrent fragment processing.** At lower query rates processing nodes cannot fully utilize all of their resources, therefore it can be useful to process the tasks corresponding to the same query concurrently. Processing each fragment completely independent from the others would require a separate sub-query state, candidate heap and pruning threshold, and therefore significantly degrade the performance. Instead, we suggest to use a small number of executors associated with each sub-query and distribute the tasks between them.

When a query $q$ is first received by the node $i$, it initiates $T_{q,i}$ task executors. Each executor initiates its own sub-query state. Similar to the previous description, a priority queue is used to order incoming fragments. In order to ensure that each executor processes fragments by increasing sequence ID and no fragments are left behind, the priority queue and fragment sequence counter are shared between the executors. We illustrate this in Fig. D.3(d).

The executors associated with the same query may share the pruning threshold variable and/or the candidate heap. Apart from the experiments presented in the next section, we have evaluated no-share policy against threshold-only and heap-and-threshold. No-share results in a lower performance as pruning efficiency goes down. With a shared candidate heap, synchronized inserts into the heap slow down processing. Our method of choice, threshold-only, is relatively cheap, since $v$ can be marked as volatile and updated by a synchronized setIfGreater($v'$) only when $v' > v$.

Since the candidate heaps are not shared, for the last sub-query, they have to be combined in order to extract the top-$k$ results. This is done by processing the sub-query as long as there are more fragments. The first executor to finish is then chosen as a heap-merger and a heap-queue is associated with the query. Each of the remaining executors, prior to finishing, inserts its candidate heap into the queue. Heaps are then taken by the merger-executor and combined with its own candidate heap. When all of the executor heaps are merged, the final $k$ results are sorted and sent back to the broker.

**Estimation of executor number.** $T_{q,i}$ can be calculated using Eq. (D.4). First, we introduce a system defined maximum number of executors per query $T_{\max}$ and divide it by the number of queries currently running on this node $Q_{\mathrm{now},i}$ plus one. Additionally, if $N_q$ is too small, the corresponding number of executors should also be smaller. Therefore, we introduce a tunable minimum number of fragments per task, $N_{\mathrm{minpt}}$. The number of executors assigned to a query is therefore the smallest of the two estimates, but not smaller than 1.

$$T_{q,i} = \max(\min(\lfloor \frac{T_{\max}}{Q_{\mathrm{now},i}+1} \rfloor, \lfloor \frac{N_q}{N_{\mathrm{minpt}}} \rfloor), 1) \tag{D.4}$$

**Multi-stage fragment processing.** As mentioned previously, each task consists of three stages (decompression, processing and compression), but the dependency in fragment execution lies only in the second stage. As an architecture design choice we decode and decompress incoming packages by small executor tasks outside of the query processing executors. Therefore, as incoming fragments enter the priority queue, they are already

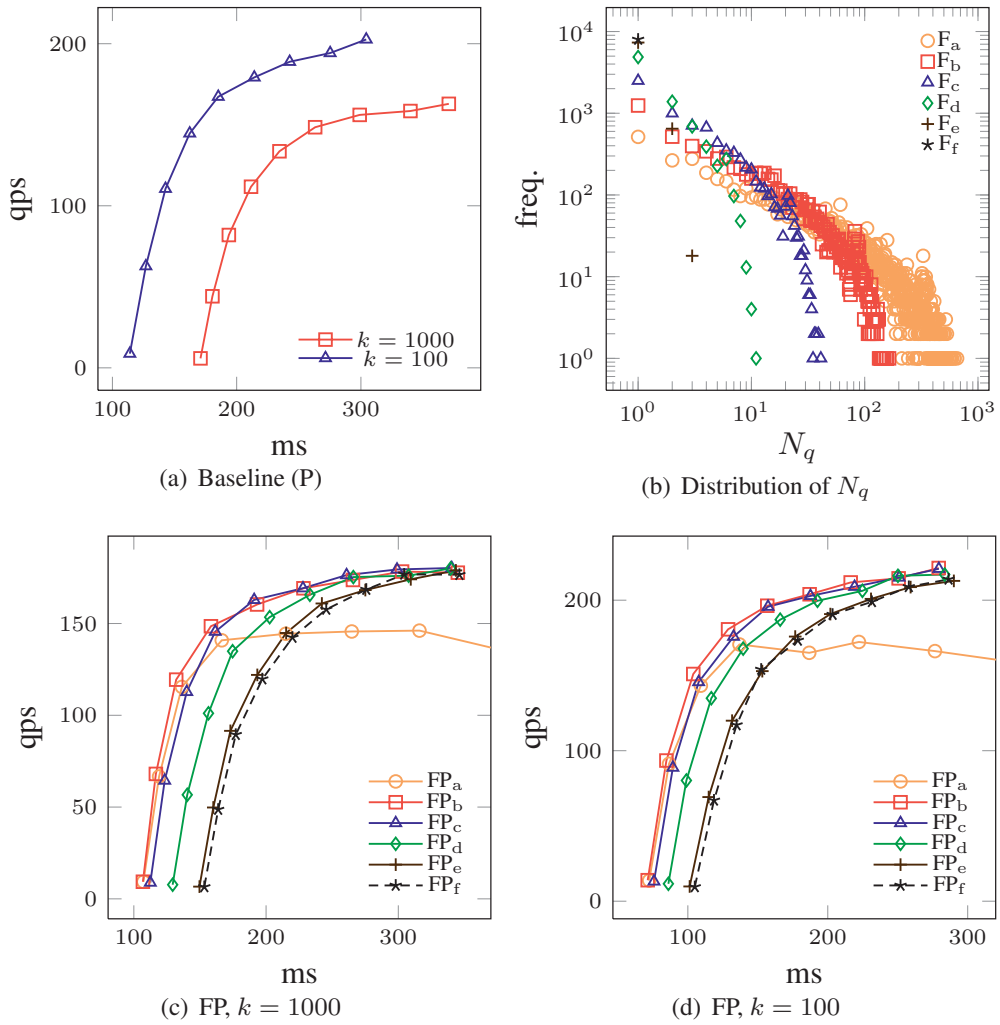
decompressed and ready for execution. This simplifies the implementation of the methods and reduces the amount of work done by the query executor. Separate post-processing and compression of outgoing data could be done by an additional executor or as a number of small executor tasks. However, the improvement is achieved only when there are many idle CPU cores. Otherwise, it only increases the overhead. For this reason, the results presented in the next section exclude separate fragment post-processing. Finally, we separate the query itself from accumulator transfer. Instead of sending a query bundle to the first node in the route, the broker multi-casts it to all of the query nodes. Therefore, each node can parse the query itself, access the lexicon and initiate the query executor(s) prior to receiving any accumulator data.

## D.5 Experiments

In this section we evaluate the performance of three query processing methods: the baseline method (P) described in Section D.3, the non-concurrent fragment pipeline (FP) described in Section D.4.1 and concurrent fragment pipeline (CFP) described in Section D.4.2. Further, both FP and CFP apply the ideas described in the multi-stage fragment processing part of the last section.

For the experiments we use the 426GB TREC GOV2 corpus. With both stemming and stop-word removal applied, the 9.4GB distributed index contains 15.4 mil. unique terms, 25.2 mil. documents, 4.7 bil. pointers and 16.3 bil. tokens. For query processing we use the Okapi BM-25 model. The performance is evaluated using the first 10 000 queries from the Terabyte Track 05 Efficiency Topics that match at least one indexed term, where the first 2 000 queries are used for warm-up and the next 8 000 (evaluation set) to measure the performance. Among the 8 000 test queries, the average query has 2.9 terms and 2.44 sub-queries. For the experiments we use a 9 node cluster interconnected with a Gigabit network. Each node has two 2.0GHz Quad-Core CPUs, 8GB memory and a SATA disk. Our framework is implemented in Java and uses Java NIO and Netty 3.2.3 for fast disk access and network transfer. For disk access we use 16KB blocks. Finally, we use the default Linux disk-cache policy, but drop the cache before each run. Every experiment is repeated twice and the average value is reported.

**Baseline.** Fig. D.4(a) illustrates the average throughput and query latency of the baseline method. Marks on each plot correspond to query multiprogramming levels {1, 8, 16, 24, 32, 40, 56, 64}, which are the maximum number of queries concurrently executing in the cluster. We stop at 64 concurrent queries (cq), which corresponds to the total number of processor cores held by the processing nodes. As the figure shows, for $k = 1000$ (i.e. when the result set is restricted to top-1000) the shortest average latency (1cq) is about 170ms, which corresponds to 5.85 queries per second (qps). As we increase the query load, both throughput and latency increase. Over the time, due to to a limited amount of resources and increasing load, the increase in latency dominates over the increase in throughput. The highest throughput is reached at 64cq, 162qps corresponding to 370ms. $k = 100$ has similar results. However, as the $k$-th best candidate score is used to calculate $v$, smaller $k$ decreases the amount of data to be read, processed and transferred, and therefore improves




(a) Baseline (P)

(b) Distribution of $N_q$

(c) FP, $k = 1000$

(d) FP, $k = 100$

Figure D.4: (a) Baseline performance. (b) Distribution of $N_q$ values in the evaluation set. (c)-(d) Performance of FP. In (b)-(d), different plots correspond to different values of $F$: a - 32 768, b - 131 072, c - 524 288, d - 2 097 152, e - 8 388 608, f - 33 554 432.

the performance. For $k = 100$, the shortest average latency is about 114ms (8.75qps) and the highest throughput is 203qps, reached at 304ms (64cq).

**Fragment pipeline.** Fig. D.4(b) illustrates the distribution of $N_q$ values in the evaluation set. Different plots correspond to different values of $F$. Here we use $F_a = 128^2 \times 2 = 32768$ as a baseline and increment the value by 4 to calculated $F_{b..f}$. As the figure shows, with $F_a$, the $N_q$ lies within [1, 654]. With $F_b$, $F_c$ and $F_d$ the ranges correspond to [1, 164], [1, 41] and [1, 11]. The of frequency of values changes also towards the smaller values. With $F_a$, $N_q = 1$ occurs 514 times. This corresponds to 1243 times with $F_b$, 2499 times with $F_c$, 4871 with $F_d$. With $F_e$, the only value of $N_q$ is 1.

Fig. D.4(c)-D.4(d) demonstrate the performance of FP. Both figures show that the shortest query latency decreases with $F$. However, with $F_a$ the method reaches a starvation point at 24cq, caused by a too large number of network messages and a significantly large process-

ing overhead due to fragmentation. Further, the results show that the difference between $F_b$ and $F_c$, and between $F_e$ and $F_f$ is less significant. At the same time, e.g., $F_b$ has a slightly shorter latency at 1cq than $F_c$, but it reaches a slightly smaller maximum throughput at 64cq. For $k = 1000$ $F_b$ reaches the maximum throughput at 56cq. At 64cq it is outperformed by the other methods. These results show that FP can significantly improve the performance at lower multiprogramming levels. For higher levels (64cq and above), the overhead from fragmentation degrades the performance.

Fig. D.5 shows the average number of read data blocks, decompressed chunks, created and transferred accumulators and sent messages for queries in the evaluation set. As pruning efficiency degrades with smaller fragments, the figure shows a small increase in the measured numbers for blocks, chunks and accumulators for FP with $F_b$ ($\mathrm{FP_b}$) and $F_c$ ($\mathrm{FP_c}$). At the same time, the number of network messages per query (except the result message) increases from 2.44 with P to 49 with $\mathrm{FP_c}$.

**Concurrent fragment pipeline.** Fig. D.6(a) illustrates the performance of CFP with one query at a time (1cq), $F_\mathrm{b}$ and varied $N_\mathrm{minpt}$. We use $T_\mathrm{max} = 8$, since each node has 8 CPU cores. For both $k = 100$ and $k = 1000$, the shortest latency is observed at $N_\mathrm{minpt} = 2$ or 3. For $N_\mathrm{minpt} = 1$ the improvement is limited due to decreased pruning efficiency and increased synchronization overhead. Fig. D.5 shows that the amount of read, decompressed and transferred data increases with smaller $F$. In the figure, $N_\mathrm{minpt}$ is indicated by the second subscript. At the same time, the amount of work that can be done concurrently increases along with $N_\mathrm{minpt}$. For $N_\mathrm{minpt} = 2$ and 3, the trade-off ratio between parallelism and overhead is therefore optimal. Similar results were observed for $F_c$ and $F_d$.

Fig. D.6(b)-D.6(c) demonstrate the performance of CFP with varied fragment size. At low multiprogramming levels the difference between $F_b$ and $F_c$ is quite significant and it is clear that smaller fragment sizes speed-up query processing. However, Fig. D.5 shows a significant increase in the processed data (1cq), which decreases with increasing multiprogramming. Despite this, at higher levels intra-query concurrency starts to degrade the performance. Fig. D.6(d)-D.6(f) show a comparison between CFP and the corresponding FP runs. The figure shows that the maximum throughput decreases for both $F_a$, $F_b$ and $F_c$, however while $\mathrm{CFP_{b3}}$ significantly reduces the latency compared to $\mathrm{FP_3}$, $\mathrm{CFP_{d3}}$ only degrades the performance of $\mathrm{FP_d}$. The last observation can be explained by a small number of fragments per-query ($F_\mathrm{d}$ gives $N_q \in [1, 11]$) and a relative high overhead cost.

The final comparison between $\mathrm{CFP_{b3}}$ and the baseline method is illustrated in Fig. D.7(a)-D.7(b). As the results show, at 1cq the query latency is 43.9ms for $k = 100$ and 64.4ms for $k = 1000$. This corresponds to a latency decrease by 2.6 times. At the same time, the maximum throughput has increased by 6.6% ($\mathrm{CFP_{b3}}$, $k = 100$, 56cq, 216.2 qps at 249.0ms) to 7.4% ($k = 1000$, 64cq, 174.9qps at 349.3ms). The most important, CFP allows to reach the same throughput as the baseline method at a half of the latency and using a lower multiprogramming level, which satisfies our main objective. For example, for $k = 100$ we reach 190qps at 159ms (32cq) compared to 275ms (56cq) with the baseline method. At higher levels ($\gg$64cq) CFP might be outperformed by P. Therefore, in order to optimize the total performance, a practical search engine could switch between P, FP and CFP depending on the query load.

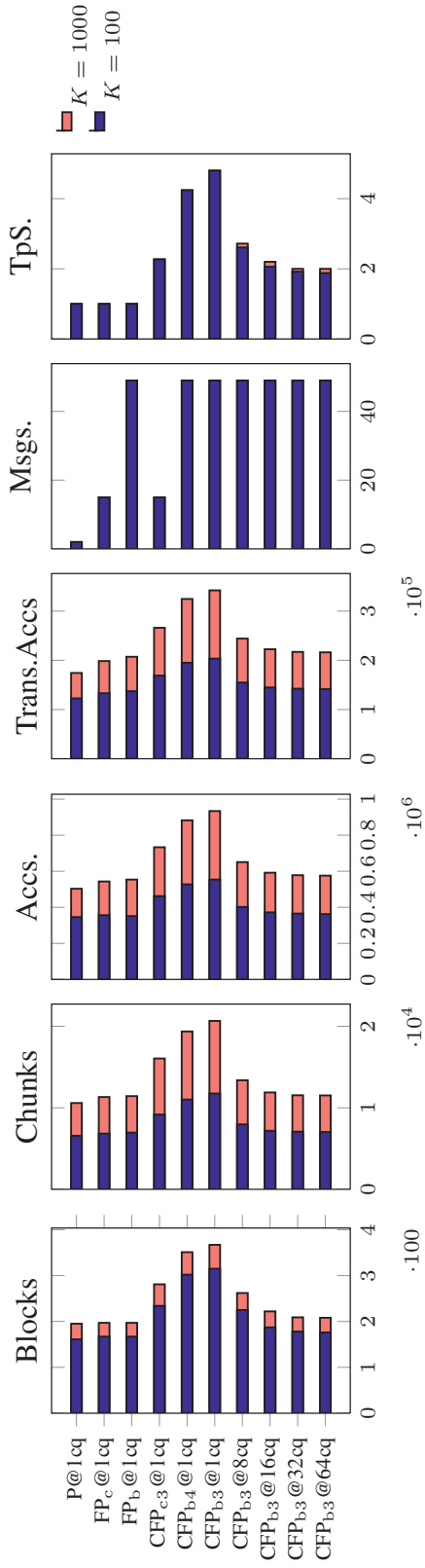



Figure D.5: Average number of processed entities per query and tasks per sub-query (TpS) observed at the given multiprogramming level.

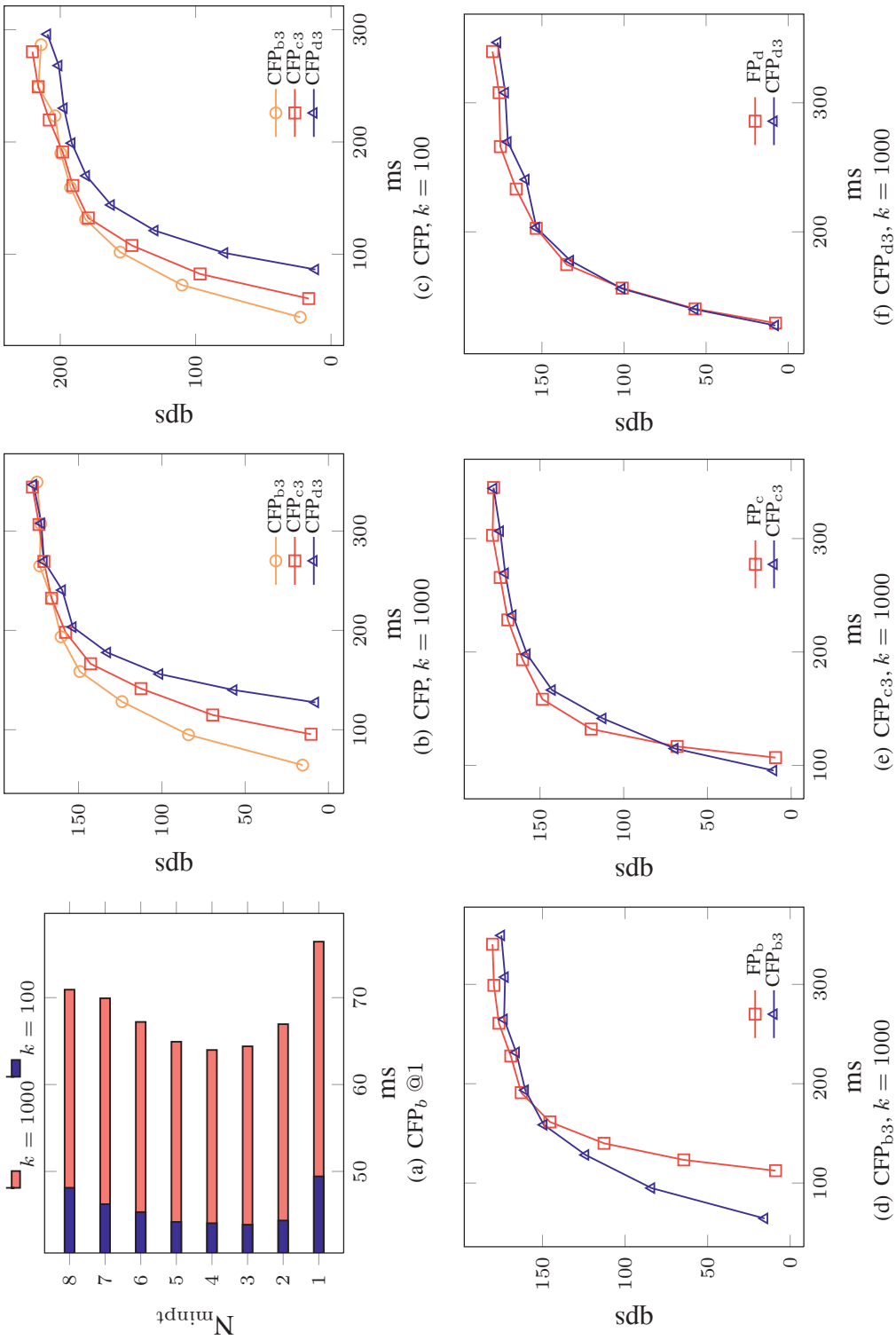
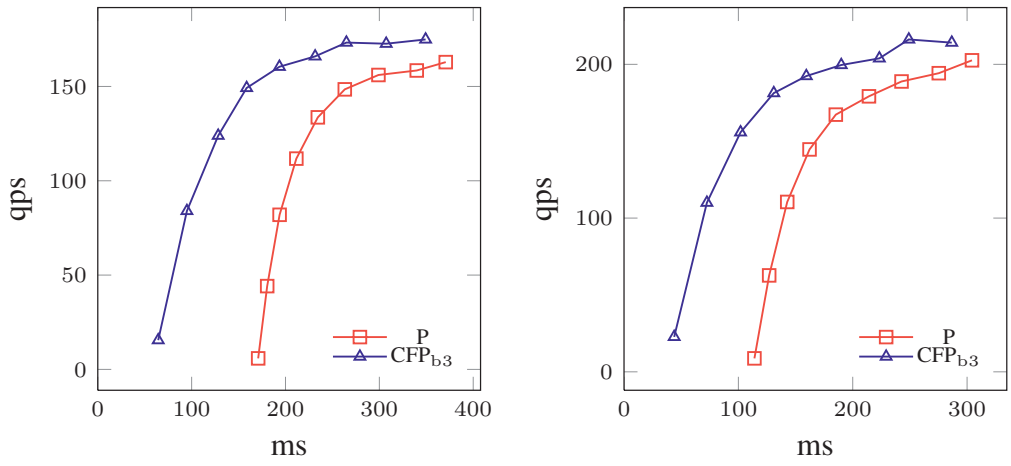



Figure D.6: Throughput and latency with the concurrent fragment pipeline (CFP).




(a) CFP$_{b3}$/P, $k = 1000$ (b) CFP$_{b3}$/P, $k = 100$

Figure D.7: Throughput and latency with the concurrent fragment pipeline (CFP). Final results.

## D.6 Conclusion and further work

In this work we have presented an efficient extension of the pipelined query processing that exploits intra-query concurrency and significantly improves query latency. Our results indicate more than 2.6 times latency improvement on the single-query case. For a general case, we are able to achieve similar throughput with almost half of the latency. Further work can be done in several directions. First, due to the assignment strategy, posting lists stored at the first few nodes are relatively short. Therefore, a hybrid combination between pipelined and non-pipelined execution, where the first nodes perform only fetching and transfer of the posting data to a node later in the query route, can significantly improve the performance. However, this technique requires careful load-balancing. A dynamic load balancing strategy can be explored as the second direction of the future work. Third, the index size and the number of nodes used in our experiments are relatively small. In the future, we plan to evaluate our method on a larger document collection and/or a larger cluster. Finally, we can also think of an efficient combination of the pipelined query execution with impact-ordered lists and bulk-synchronous processing similar to the methods presented by Marin *et al.* [9, 10].

**Acknowledgments.** This work was supported by the Information Access Disruptions Centre (http://iad-centre.no) funded by the Research Council of Norway and the Norwegian University of Science and Technology. The authors thank João B. Rocha-Junior for the useful advices and comments on the paper.

## D.7 References

[1] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, 2001.

[2] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[3] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, 2011.

[4] E. Feuerstein, M. Marin, M. Mizrahi, V. Gil-Costa, and R. Baeza-Yates. Two-dimensional distributed inverted files. In *SPIRE*, 2009.

[5] S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *WISE*, 2010.

[6] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.

[7] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 2005.

[8] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale*, 2007.

[9] M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *CIKM*, 2007.

[10] M. Marin, V. Gil-Costa, C. Bonacic, R. Baeza-Yates, and I. Scherson. Sync/async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 2010.

[11] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, 2006.

[12] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 2007.

[13] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL*, 1998.

[14] T. Strohman and W. Croft. Efficient document retrieval in main memory. In *SIGIR*, 2007.

[15] A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in shared nothing text document information retrieval systems. *The VLDB Journal*, 1993.

[16] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 1995.

[17] W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, University of Melbourne, 2007.

[18] W. Xi, O. Sornil, M. Luo, and E. Fox. Hybrid partition inverted files: Experimental validation. In *ECDL*, 2002.

[19] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.

[20] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. *Paral. and Dist. Proc. Symp., Int.*, 2007.

Appendix E

# Paper B.III: Improving Dynamic Index Pruning via Linear Programming

Simon Jonassen and B. Barla Cambazoglu.
*under submission / in progress*

**Abstract:** Dynamic index pruning techniques are commonly used to speed up query processing in web search engines. In this work, we propose a linear programming technique that can further improve the performance of the state-of-the-art dynamic index pruning techniques. The experiments we conducted show that the proposed technique achieves reduction in terms of the disk access, index decompression, and scoring costs compared to the well-known Max-Score technique.

# E.1 Introduction

In search engines, queries are processed on an inverted index, which maintains an inverted list for each term in the collection vocabulary. An inverted list contains entries called postings, each of which keeping some information about a document in which the term associated with the list appears (e.g., document id, term frequency). Typically, the postings in a list are sorted in increasing order of document ids [4].

Queries are processed by iterating over the lists associated with the query terms and accumulating score contributions for the documents encountered in the postings. For a given query $q = \{t_1, t_2, \ldots, t_L\}$ of $L$ terms, the relevance between $q$ and a certain document $d$ is computed as $s(d, q) = \sum_{i=1}^{L} w(d, t_i)$, where $w$ is a function (e.g., BM25) indicating the degree of relevance between a document and a query term. The scores are typically computed following the document-at-a-time strategy, where the final score for the document that has the lowest id among the unprocessed documents in the lists is fully computed before any other document is scored. The documents with the highest $k$ scores are maintained in a min-heap as the computation proceeds, and the final top $k$ set is returned to the user.

Dynamic index pruning is a well-known technique used to speed up the above-mentioned type of query processing operations. By this technique, some redundant operations that will have no impact on the final top $k$ result set can be avoided, i.e., certain portions of the lists can be skipped and some score computations are only partially carried out. The most well-known pruning techniques are Max-Score [5, 9] and WAND [1]. Both techniques rely on intelligent ordering of inverted lists during query processing and computation of some upper bounds on scores to early terminate score computations for some documents. If we omit specific optimizations [3, 8], the main difference between the two techniques is in the order in which the inverted lists are processed.

In this work, we propose a linear programming (LP) approach that can further improve the performance of the Max-Score and WAND techniques.[1] The previous works, so far, compute score upper bounds associated with a single inverted list [6] or blocks in the lists [3]. Herein, we present a solution that computes tighter score bounds for *subsets of query terms* by using the maximum scores observed for previously issued queries. In particular, we obtain the maximum scores for single terms and some frequent query term pairs. We then use an LP solver to estimate the maximum scores for subsets of a given query. The resulting scores are used as score upper bounds by the Max-Score technique. We show that, for queries with three to six terms, this approach leads to significant savings in the amount of data that is read and decompressed as well as the number of score updates.

[1] Although our technique is applicable to both techniques, due to space limitations, we only concentrate on Max-Score. A good description of the two techniques and a performance comparison between them can be found in [4] and [7].

## E.2 The Max-Score Technique

Let $\tilde{s}$ denote the $k$-th highest score observed so far (i.e., the head of the min-heap), $\hat{s}_i$ denote the maximum possible score contribution from term $t_i$, and $a_i$ denote the sum of the maximum possible score contributions for the terms in set $\{t_i, t_{i+1}, \ldots, t_L\}$, i.e., $a_i = \sum_{j=i}^{L} \hat{s}_j$. In the Max-Score technique, the terms (i.e., lists) are always processed in decreasing order of the $\hat{s}$ values. Before scoring a document, the query terms are divided into two subsets: $\{t_1, t_2, \ldots, t_{r-1}\}$ (*required*) and $\{t_r, t_{r+1}, \ldots, t_L\}$ (*optional*), where $r$ is the smallest integer such that $a_r < \tilde{s}$. Initially, all terms are placed in the required set. As more documents are scored and $\tilde{s}$ increases, the terms are moved into the optional set. Since $a_r < \tilde{s}$, the optional terms cannot create new candidate documents. Hence, the selection of candidates is based only on the required set. Once a candidate is selected, the lists are considered from $t_1$ to $t_L$, and the pointers on the optional lists are advanced with skip operations. The scoring of a candidate can be terminated prior to considering $t_i$ if the current score plus the maximum possible score is below the score of the current $k$-th best candidate, i.e., $s + a_i < \tilde{s}$.

## E.3 Linear Programming Solution

Our approach is to compute tighter cumulative score upper bounds using the maximum scores assigned to subsets of query terms issued to the search engine in the past. To this end, we consider each subquery $\bar{q}_i = \{t_i, t_{i+1} \ldots, t_L\}$ as subject to the LP problem of finding $\ell(\bar{q}) = \max \sum_{t_j \in \bar{q}} x_j$ such that the conditions in Eqs. (E.1) and (E.2) are satisfied.

$$\sum_{t_j \in q'} x_j \leq \hat{s}_{q'}, \forall q', \text{ s.t. } q' \in Q \text{ and } q' \in \bar{q}, \tag{E.1}$$

$$x_j \geq 0, \forall t_j, \text{ s.t. } t_j \in \bar{q}. \tag{E.2}$$

Here, $\hat{s}_{q'}$ denotes the maximum score for a query $q'$ in a set $Q$ of past queries and $x_j$ is a real-valued variable associated with $t_j \in \bar{q}$ used by the optimization problem.

Herein, we also consider two general cases corresponding to the conjunctive (AND) and disjunctive (OR) modes of query processing. For the AND mode, $\ell(\bar{q}_i)$ is a safe approximation to $a_i$. Hence, the modified version of the pruning algorithm requires computing $\ell(\bar{q}_2)$ to $\ell(\bar{q}_L)$ to be used as $a_2$ to $a_L$. Since $a_1$ is never used, the most expensive call to the LP solver is avoided, and the total number of calls is $O(L)$. For the OR mode, a safe approximation is $a_i = \max(\ell(q'_i))$, where $q'_i$ is any subquery of $\bar{q}_i$. In practice, dynamic programming can be used to reduce the number of calls to the LP solver. However, the total number of calls will be $O(2^L)$.

## E.4 Experimental Results

We build and index on 20 million pages sampled from the UK domain and use a commercial query log. In the training phase, we find the maximum scores for all terms in the

index and also compute the maximum scores for three million most frequent term pairs. In the evaluation phase, we use the next 100K queries from the query log. All queries are normalized via traditional techniques and those with no matching results are ignored. 64.9% of the evaluated queries contain one or two terms, 34.5% contain three to six terms, and 0.5% contain seven or more terms. Because of the high LP overhead and low performance gains, we do not report the results for queries with more than six terms. For one- and two-term queries, our approach is identical to Max-Score.

The implementation of the index and Max-Score is similar to those in [5] and the LP solver is similar to that in [2]. Table E.1 shows the performance of the Max-Score and the relative improvements achieved by our technique ($k \in \{1, 10\}$) for the number of blocks (1 KB) read from the index, decompressed index chunks (each with 128 document ids, frequencies, or skip-pointers), and the number of computed scores.

The results indicate that, for the AND mode, our technique slightly improves the I/O and decompression while, for the OR mode, it significantly reduces the number of score computations. The improvement decreases with increasing $k$ (it is marginal for $k \in \{100, 1000\}$). We note that, for short queries, the overhead due to LP computations is negligible compared to the cost of query processing. However, for long queries (e.g., $L > 6$), the LP solver may be a bottleneck, especially if the index size is very small.

## E.5 Conclusions

We presented a linear programming approach to improve the performance of dynamic index pruning techniques. Our approach can compute relatively tighter score upper bounds for a query, making use of the maximum scores observed for subsets of the query terms issued in the past. Despite the fact that the observed savings are not very high, we believe that the proposed approach can lead to some financial gains in case of very large indexes built on billions of documents.

## E.6 References

[1] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[2] B. B. Cambazoglu, E. Varol, E. Kayaaslan, C. Aykanat, and R. Baeza-Yates. Query forwarding in geographically distributed search engines. In *SIGIR*, 2010.

[3] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, 2011.

[4] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, 4(12), 2011.

Table E.1: Percent improvements ($\%I$) achieved over Max-Score with respect to the number (x1000) of blocks read ($B$), chunks decompressed ($C$), and scores computed ($S$) by Max-Score ($L$ denotes the query length)

| | Blocks read | | | | | | | | Chunks decompressed | | | | | | | | Scores computed | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AND | | | | OR | | | | AND | | | | OR | | | | AND | | | | OR | | | |
| | $k=1$ | | $k=10$ | | $k=1$ | | $k=10$ | | $k=1$ | | $k=10$ | | $k=1$ | | $k=10$ | | $k=1$ | | $k=10$ | | $k=1$ | | $k=10$ | |
| $L$ | $B$ | $\%I$ | $B$ | $\%I$ | $B$ | $\%I$ | $B$ | $\%I$ | $C$ | $\%I$ | $C$ | $\%I$ | $C$ | $\%I$ | $C$ | $\%I$ | $S$ | $\%I$ | $S$ | $\%I$ | $S$ | $\%I$ | $S$ | $\%I$ |
| 3 | 0.6 | 3.0 | 0.9 | 0.4 | 0.7 | 3.3 | 1.0 | 0.6 | 3.9 | 6.2 | 5.4 | 2.6 | 4.1 | 6.5 | 6.2 | 2.8 | 94 | -1.5 | 87 | -1.2 | 115 | 2.0 | 147 | 3.7 |
| 4 | 0.7 | 3.2 | 1.0 | 0.9 | 0.8 | 4.0 | 1.3 | 1.3 | 4.1 | 5.2 | 5.5 | 2.4 | 4.7 | 7.2 | 7.9 | 4.2 | 68 | -1.8 | 65 | -0.6 | 109 | 8.1 | 167 | 9.0 |
| 5 | 0.7 | 2.6 | 0.9 | 0.8 | 0.9 | 4.4 | 1.6 | 1.5 | 3.6 | 4.1 | 4.7 | 1.6 | 5.4 | 8.3 | 9.6 | 4.6 | 38 | -1.9 | 37 | -0.3 | 120 | 16.5 | 205 | 11.4 |
| 6 | 0.6 | 2.6 | 0.8 | 0.8 | 1.1 | 4.8 | 1.9 | 1.8 | 3.1 | 4.0 | 3.9 | 1.5 | 6.5 | 9.8 | 11.6 | 5.2 | 26 | -1.5 | 26 | -0.1 | 154 | 20.0 | 263 | 12.2 |

[5] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.

[6] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM Trans. Inf. Syst.*, 29(4), 2011.

[7] D. Shan, S. Ding, J. He, H. Yan, and X. Li. Optimized top-k processing with global page scores on block-max indexes. In *WSDM*, 2012.

[8] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, 2005.

[9] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31, 1995.

Appendix F

# Paper C.I: Modeling Static Caching in Web Search Engines

Ricardo Baeza-Yates and Simon Jonassen.
*Appeared at the 34th European Conference on Information Retrieval (ECIR), Barcelona, Spain, April 2012.*

**Abstract:** In this paper we model a two-level cache of a Web search engine, such that given memory resources, we find the optimal split fraction to allocate for each cache, results and index. The final result is very simple and implies to compute just five parameters that depend on the input data and the performance of the search engine. The model is validated through extensive experimental results and is motivated on capacity planning and the overall optimization of the search architecture.

# F.1 Introduction

Web search engines are crucial to find information among more than 180 million Web sites active at the end of 2011[1], and users expect to rapidly find good information. In addition, the searchable Web becomes larger and larger, with more than 50 billion static pages to index, and evaluating queries requires the processing of larger and larger amounts of data each day. In such a setting, to achieve a fast response time and to increase the query throughput, using a specialized cache in main memory is crucial.

The primary use of a cache memory is to speedup computation by exploiting frequently or recently used data. A secondary but also important use of a cache is to hold pre-computed answers. Caching can be applied at different levels with increasing response latencies or processing requirements. For example, the different levels may correspond to the main memory, the disk, or resources in a local or a wide area network. In the Web, caching can be at the client side, the server side, or in intermediate locations such as a Web proxy [14].

The cache can be static or dynamic. A static cache is based on historical information and can be periodically updated off-line. If the item that we are looking for is found in the cache, we say that we have a hit, otherwise we say that we have a miss. On the other hand, a dynamic cache replaces entries according to the sequence of requests that it receives. When a new request arrives, the cache system has to decide whether to evict some entry from the cache in the case of a cache miss. These decisions are based on a cache policy, and several different policies have been studied in the past.

In a search engine there are two possible ways to use a cache memory:

**Caching results:** As the engine returns results to a particular query, it may decide to store these results to resolve future queries. This cache needs to be periodically refreshed.

**Caching index term lists:** As the search engine evaluates a particular query, it may decide to store in memory the inverted lists of the involved query terms. As usually the whole index does not fit in memory, the engine has to select a subset to keep in memory and speed up the processing of queries.

For designing an efficient caching architecture for web search engines there are many trade-offs to consider. For instance, returning an answer to a query already existing in the cache is much more efficient than computing the answer using cached inverted lists. On the other hand, previously unseen queries occur more often than previously unseen terms, implying a higher miss rate for cached results. Caching of inverted lists has additional challenges. As inverted lists have variable size, caching them dynamically is not very efficient, due to the complexity involved (both in efficiency and use of space) and the skewed distribution of the query stream. Neither is static caching of inverted lists a trivial task: when deciding which terms to cache one faces the trade-off between frequently queried terms and terms with small inverted lists that are space efficient. Here we use the algorithm proposed by Baeza-Yates *et al.* in [2]. In that paper it is also shown that in spite that the query distribution changes and there are query bursts, the overall distribution changes so little that the static cache can be precomputed every day without problems.

[1] According to Netcraft, January 2012.

This paper also leaves open the problem on how to model the optimal split of the cache between results and inverted lists of the index, which we tackle here.

In fact, in this paper we model the design of the two cache level explained before, showing that the optimal way to split a static cache depends in a few parameters coming from the query and text distribution as well as on the exact search architecture (e.g. centralized or distributed). We validate our model experimentally, showing that a simple function predicts a good splitting point. In spite that cache memory might not be expensive, using this resource well does change the Web search engine efficiency. Hence, this result is one component in a complete performance model of a Web search engine, to do capacity planning and fine tuning of a given Web search architecture in an industrial setting.

The remainder of this paper is organized as follows. Section F.2 covers related work while Section F.3 shows the characteristics of the data that we used to find the model as well as perform the experimental validation. Section F.4 presents our analytical model while Section F.5 presents the experimental results. We end with some conclusions in Section F.6.

## F.2 Related Work

Query logs constitute a valuable source of information for evaluating the effectiveness of caching systems. As first noted by Xie and O'Hallaron [18], many popular queries are shared by different users. This level of sharing justifies the choice of a server-side caching system in Web search engines. One of the first papers on exploiting user query history was proposed by Raghavan and Sever [15]. Although their technique is not properly caching, they suggest using a *query base*, built upon a set of persistent "optimal" queries submitted in the past, in order to improve the retrieval effectiveness for similar future queries. That is, this is a kind of static result cache. Later, Markatos [10] shows the existence of temporal locality in queries, and compares the performance of different caching policies.

Based on the observations of Markatos, Lempel and Moran proposed an improved caching policy, Probabilistic Driven Caching, based on the estimation of the probability distribution of all possible queries submitted to a search engine [8]. Fagni *et al.* follow Markatos' work by showing that combining static and dynamic caching policies together with an adaptive prefetching policy achieves a high hit QTFDF [6].

As search engines are hierarchical system, some researchers have explored multi-level architectures. Saraiva *et al.* [16] proposes a new architecture for web search engines using a dynamic caching system with two levels, targeted to improve response time. Their architecture use an LRU policy for eviction in both levels. They find that the second-level cache can effectively reduce disk traffic, thus increasing the overall throughput. Baeza-Yates and Saint-Jean propose a three level index organization for Web search engines [5], similar to the one used in current architectures. Long and Suel propose a caching system structured according to three different levels [9]. The intermediate level contains frequently occurring pairs of terms and stores the intersections of the corresponding inverted lists. Skobeltsyn *at al.* [17] adds a pruned index after the result cache showing that this idea is not effective as inverted list caching basically serves for the same purpose.

Later, Baeza-Yates *et al.* [2, 3] explored the impact of different static and dynamic techniques for inverted list caching, introducing the QTFDF algorithm for static inverted list caching that we use here. This algorithm improves upon previous results on dynamic caching with similar ideas [9].

More recent work on search engine caching includes how to avoid cache pollution in the dynamic cache [4] and how to combine static and dynamic caching [1]. Gan and Suel improve static result caching to optimize the overall processing cost and not only the hit ratio [7], an idea also explored by Ozcan *et al.* [13] for dynamic caching. In a companion paper, Ozcan *et al.* [12] introduce a 5-level static caching architecture to improve the search engine performance.

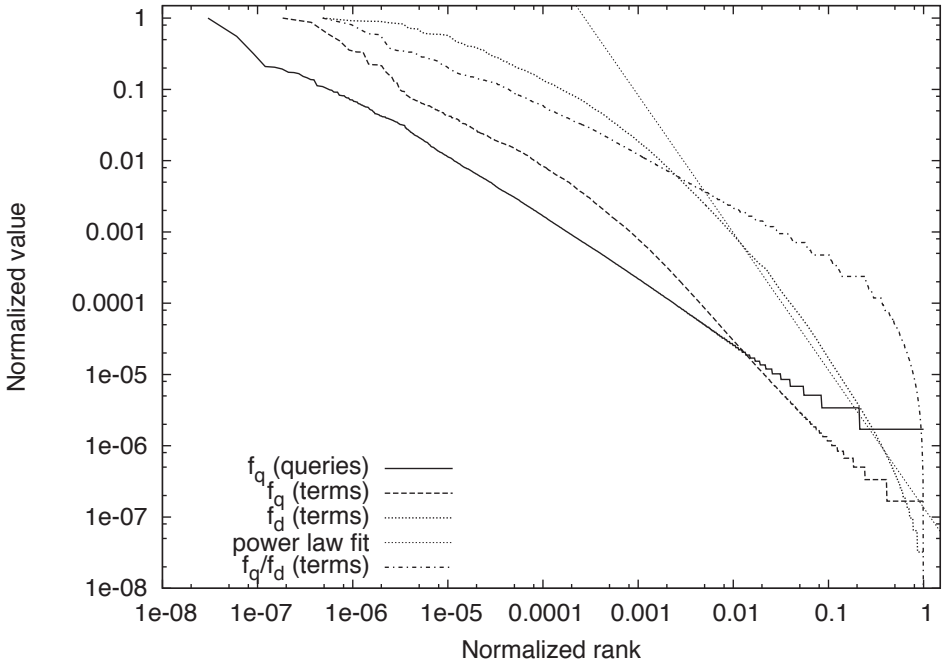



Figure F.1: Queries and terms distribution as well as the query-text term frequency ratio.

## F.3 Data Characteristics

For the experiments we use a 300GB crawl of the UK domain from 2006, which contains 37.9 million documents, and a random sample of 80.7 million queries submitted to a large search engine between April 2006 and April 2007. In both the query log and the document collection, we normalize tokens, replace punctuation characters with spaces and remove special characters. The complete query log contains 5.46 million unique terms, with 2.07 million of which appear in the document collection. For these terms we count the total number of documents (document frequency) where each of these terms appears. Finally,

we split the query log into two logs - a training log containing the first 40 million queries and a testing log for evaluation containing the remaining 40.3 million queries.

The query distribution as well as the term distribution in the queries and in the Web collection are shown in Figure F.1. In this figure we also show the distribution of the ratio of the query and text term frequencies, as this is the heuristic used in the static inverted list caching algorithm (QTFDF), which fills the cache using the inverted lists in decreasing order of this ratio. All these distributions follow a power law in their central part and hence a good approximation for those curves is $k/r^{\gamma}$, where $r$ is the item rank in decreasing order. Table F.1 shows these parameters for the different variables, where $u$ is the overall fraction of unique items, a value needed later.

Table F.1: Characteristics of the power law distributions.

| Variable | $\gamma$ | $k$ | $u$ |
|---|---|---|---|
| Queries | 0.8635 | $1.1276 \cdot 10^{-6}$ | 0.466 |
| Query terms | 1.3532 | $1.0388 \cdot 10^{-3}$ | 0.039 |
| Text terms | 1.9276 | $1.3614 \cdot 10^{-7}$ | |
| Tfq-Tfd ratio | 0.7054 | $2.0328 \cdot 10^{-4}$ | |

Notice that the $\gamma$ value for the distribution of the ratio of term frequencies is almost the ratio of the $\gamma$ values of the two distributions involved in spite that the correlation between both term distributions is only 0.4649.

## F.4 Modeling the Cache

Our static cache is very simple. We use part of the cache for results and the rest for the inverted lists of the index. Our memory cache do not need to be in the same server. That is, we have an overall amount of memory, that can be split into two parts. Usually the results cache will be closer to the client (e.g. in the front end Web server or a proxy) and the index cache could in a local (centralized case) or a remote (distributed or WAN case) search server. Therefore, our problem is how to split the memory resources in the best possible way to minimize the search time.

We first need to model the hit ratio, that is, the probability of finding a result in the cache for a query. As shown in the previous section, the query distribution follows a power law. Hence, the hit ratio curve will follow the surface under the power law. That is, if we approximate the query distribution by the function $k/r^{\gamma}$, the hit ratio function will follow a function of the form $k'r^{1-\gamma}$ reaching a limit for large $r$ as all unique queries will always be missed (in most query logs unique queries will be roughly 50% of the volume). This hit ratio curve is shown for the result cache in Figure F.2 (top) and the limit is $1-u$ from Table F.1.

The optimal split for a given query distribution is not trivial as we have two cascading caches and the behavior of the second will depend on the performance of the first. Hence, modeling this dynamic process is quite difficult. However, based in our previous work [17]

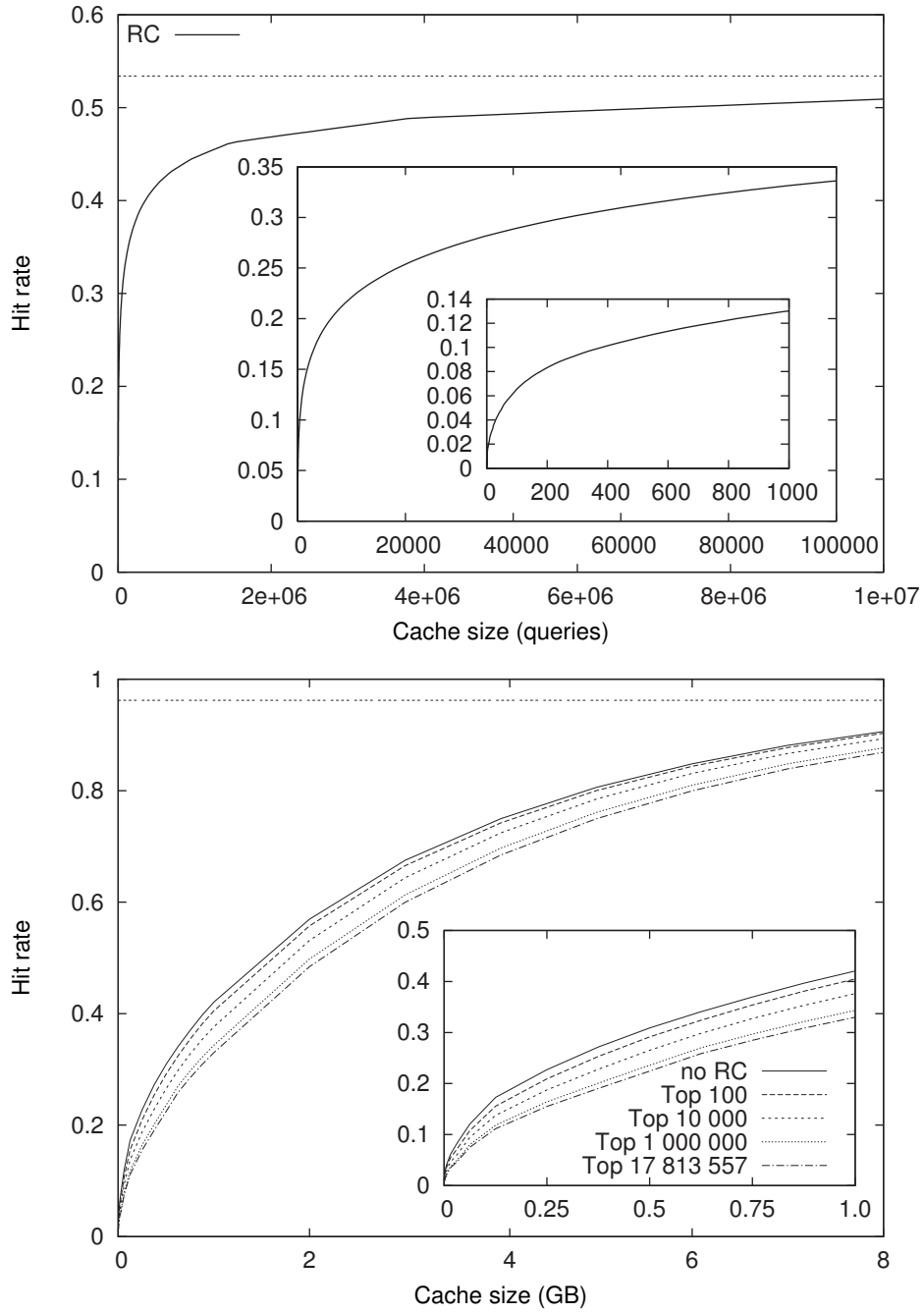



Figure F.2: Hit ratio in the results (top) and the inverted list (bottom) cache.

we notice that the query distribution after the result cache had basically the same shape as the input query distribution. We corroborate that finding, plotting in Figure F.2 (bottom) the query distribution after the result cache for different cache sizes. So an approximate model is to assume that the hit ratio curve for inverted lists is independent of the result cache. As the distribution of cached inverted lists also follows a power law, we use the same function as the hit ratio curve for the result cache, and only the parameters change.

That is, the hit ratio in both cases is modeled by

$$h(x) = \frac{k}{x^{\gamma}}$$

where $x$ is the cache fraction used. For large $x$, $h(x)$ converges to the limit $1-u$, and we force that by defining $h(1) = 1-u$ which then sets the value of $k$. As we will see later, this second approximation does not affect much the analytical results and is similar to the infinite cache case. Notice that we have already found the experimental values of $u$ and $\gamma$ in Section F.3.

Assuming that this model is a good approximation, we can use it to find the optimal split fraction for this case. Let $x$ be that optimal point and $h_R$ and $h_I$ the corresponding hit functions. Hence, the search time is given by

$$T(x) = t_R\, h_R(x) \;+\; t_I\, (h_I(1-x) - h_R(x)) \;+\; t_P\, (1 - h_I(1-x))$$

where $t_R$, $t_I$, and $t_P$ are the average response time of the search engine when the query is answered from the result cache, the index cache or has to be fully processed. Notice that for the range of interest (e.g. $x > 0.5$) and that given that in practice $h_I$ is much larger than $h_R$ as we will see in the next section, the second term is always positive.

Simplifying the previous formula and computing the partial derivative of the resultant expression with respect to $x$, we obtain that the optimal split must satisfy the following equation

$$\frac{(1-x)^{\gamma_L}}{x^{\gamma_R}} = \frac{k_L\, (t_P - t_I)(1-\gamma_L)}{k_R\, (t_I - t_R)(1-\gamma_R)}$$

which can be solved numerically. However, by expanding the left hand side, and considering the relative answer time (that is, we set $t_R = 1$) and that $t_I >> t_R$ in the right hand side, we obtain that the split point $x^*$ can be approximated by a simple function

$$x^* = \frac{1}{TRr^{1+\gamma_L/\gamma_R}}\ ,$$

where $TRr = t_P/t_I$. In the following section we validate this model.

## F.5 Experimental Validation

We performed experiments simulating both cases, the centralized and the distributed case. In the centralized case the front end Web server is directly connected to the search server. The distributed architecture implies a local front end connected to a remote server through an Internet connection. We did not consider the local area network case (e.g. the case of a cluster) because the results were basically the same as in the centralized case. For comparison purposes we used the processing times of [2], shown in Table F.2. Here we consider only the compressed case as this is what it is used in practice, but we compare our model also with the previous uncompressed results in the next section. We also consider two cases when processing the query: full evaluation (compute all possible answers) and

partial evaluation (compute the top-10k answers). The two-level cache was implemented using a fraction of the main memory available with the result cache in the front end Web server and the index cache in the search server. We do not consider the refreshing of the result cache as there are orthogonal techniques to do it, and that refreshing the stored results do not change the queries stored in the cache.

Table F.2: Ratios between the average time to evaluate a query and the average time to return cached results for the centralized and distributed cases with full or partial evaluation, and uncompressed (1Gb cache) and compressed (0.5Gb cache) index.

| System | Uncompressed | | Compressed | |
|---|---|---|---|---|
| Centralized | $TR_1^C$ | $TR_2^C$ | $TR_1'^C$ | $TR_2'^C$ |
| Full evaluation | 233 | 1760 | 707 | 1140 |
| Partial evaluation | 99 | 1626 | 493 | 798 |
| Distributed | $TR_1^D$ | $TR_2^D$ | $TR_1'^D$ | $TR_2'^D$ |
| Full evaluation | 5001 | 6528 | 5475 | 5908 |
| Partial evaluation | 4867 | 6394 | 5270 | 5575 |

We model the size of compressed inverted lists as a function of the document frequency. In our previous work [2] we used the Terrier system [11] that uses gamma encoding for document ID gaps and unary codes for frequencies. Here we use the state of the art, compressing the inverted lists with the NewPFor approach by Zhang, Long and Suel [19]. We have analyzed both techniques in the TREC GOV2 corpus and NewPFor is in general a bit better. In this method inverted lists are grouped into blocks of 128 entries and compressed together - one block of compressed $d$-gaps is followed by a block of compressed frequencies. Blocks shorter than 100 entries are compressed using VByte. In the following we use NewPFor compression which we approximate with the following function:

$$sizeB(docs) = \begin{cases} \text{if } docs < 100 \text{ then} & 1.81 + 3.697 \cdot docs \\ \text{else} & 176.42 + 2.063 \cdot docs \end{cases}$$

For the results cache we use entries of size 1264 bytes.

We consider cache memories that are a power of 2 starting with 128Mb and finishing in 8Gb, using query frequency in decreasing order to setup the results cache and using the QTFDF algorithm to setup the index cache. The setup is done with the training query log while the experimental results are done with the testing query log that were described in Section F.3. In Figure F.3 we show one example for the response time for a memory cache of 1Gb in function of the cache size used for results. In the centralized case the optimal slit is in 0.62 while in the distributed case is almost 1. In Figure F.4 we show the optimal fraction for the results cache in function of the overall cache size. As the results cache size reaches a saturation point, the optimal fraction decreases while the overall cache size increases.

We also tried a variant of the static caching algorithm by not considering terms with $f_q < A$ and then using a modified weight: $f_q / \min(f_d, B)$ with different $A$ and $B$ in the set $\{100, 1000, 10000\}$. This did improve the performance but only for queries involving very long inverted lists.

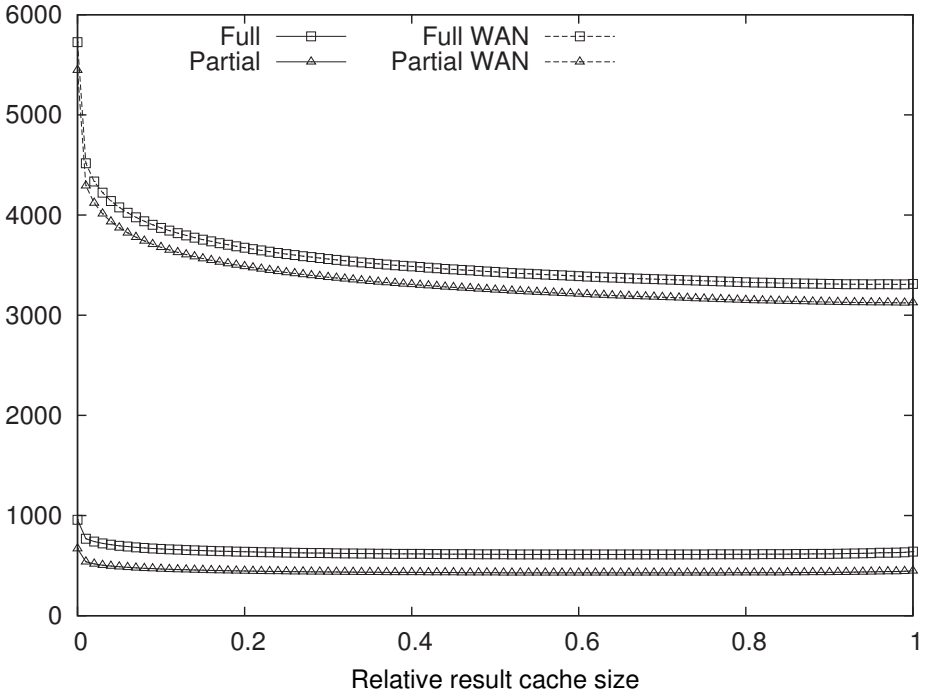



Figure F.3: Answer time performance for different splits in a cache of 1GB.

In Figure F.5 we compare our experimental results and the results of our previous work [2] with our model and our approximated solution depending on $TRr$ which is the ratio $t_P/t_I > 1$ for the different cases. The predicted optimal ratio is quite good, in particular for partial evaluation (lower data points) which is also the most realistic case in a Web search engine.

## F.6 Conclusions

We have shown that to split almost optimally a static cache, for our Web search engine, we just need to compute five parameters: the power law exponent estimation for the distribution of queries and query terms as well as for document terms, plus the average response time when answering with the index cache or when processing the whole query. Notice that this assumes that we can compute the power law of the query-document term ratio with just a division. Otherwise a sixth parameter needs to be computed.

Further work includes doing further experimental results with more query log samples, using those results to improve this model and later extend it to more complex static caching schemes [7, 12].

**Acknowledgments.** This work was done while the second author was an intern at Yahoo! Research and supported by the iAd Centre (http://iad-centre.no) funded by the Research Council of Norway and the Norwegian University of Science and Technology.

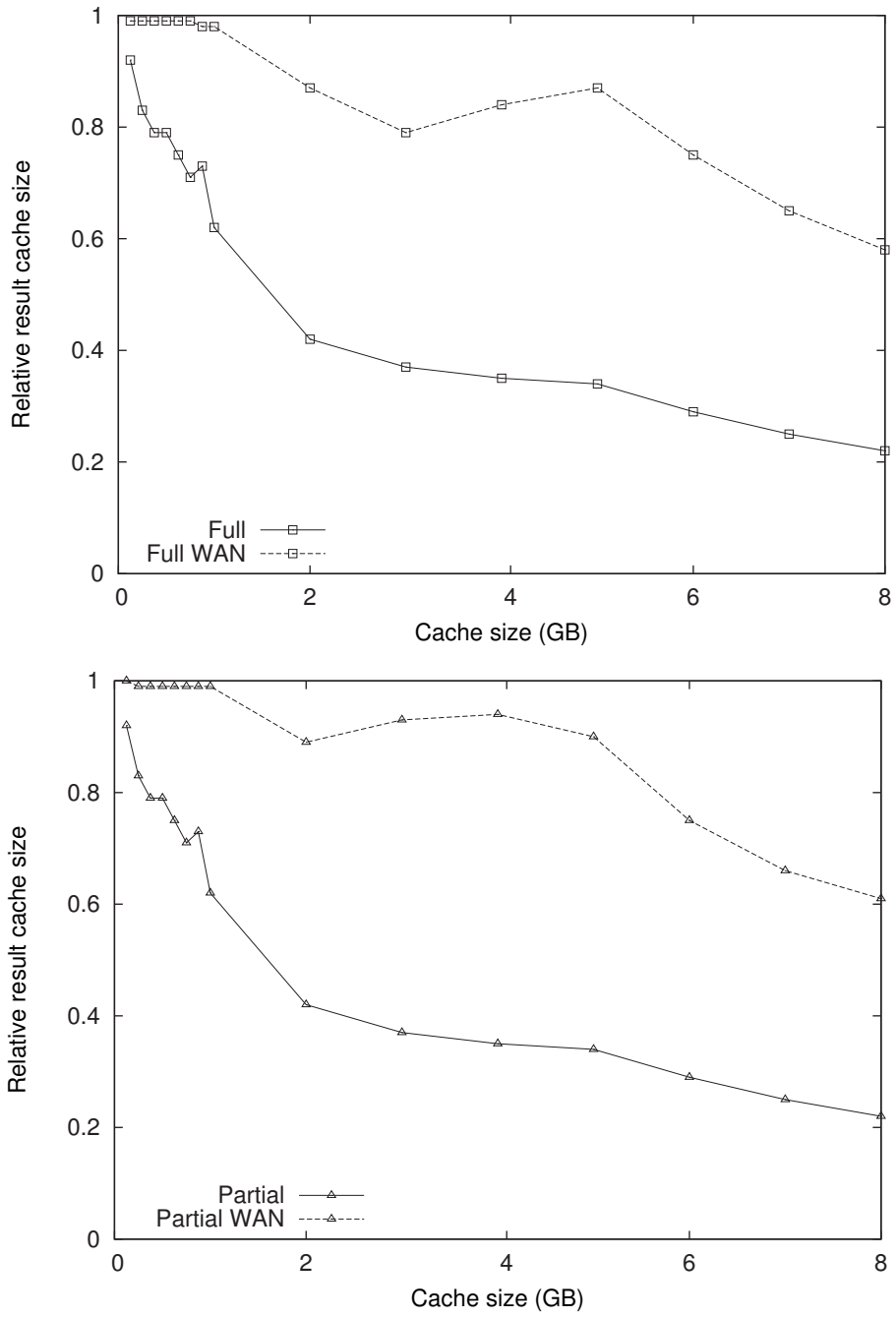



Figure F.4: Optimal split depending on the cache size for full (top) and partial (bottom) evaluation.

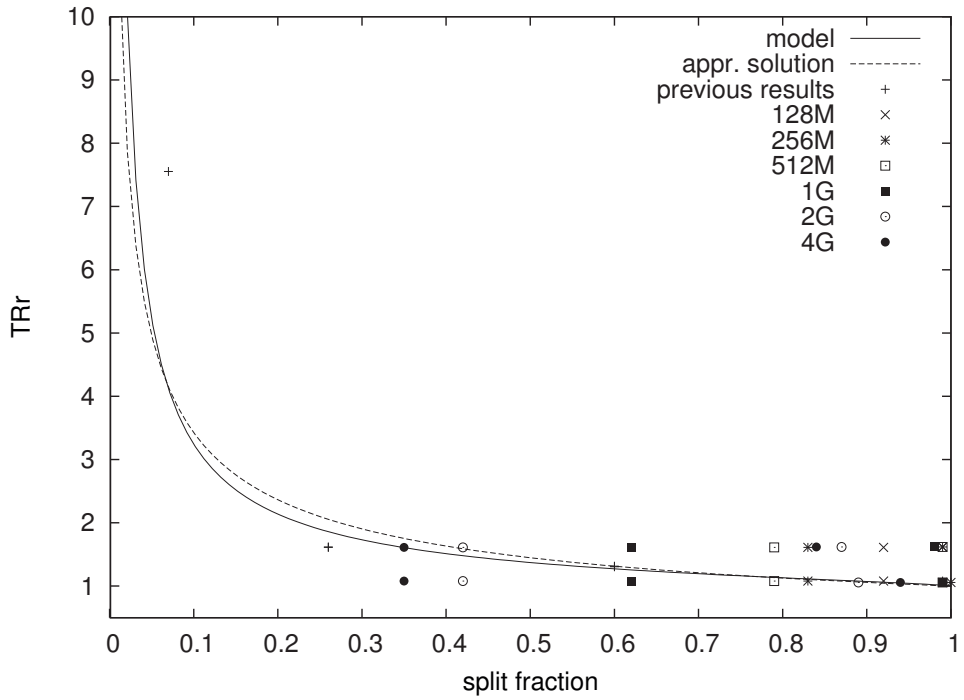



Figure F.5: Comparison of the experimental results and the model.

## F.7 References

[1] Altingovde, I., Ozcan, R., Cambazoglu, B., Ulusoy, O.: Second chance: A hybrid approach for dynamic result caching in search engines. In: ECIR 2011, pp. 510–516 (2011)

[2] Baeza-Yates, R.A., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: SIGIR. pp. 183–190 (2007)

[3] Baeza-Yates, R.A., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: Design trade-offs for search engine caching. TWEB 2(4) (2008)

[4] Baeza-Yates, R.A., Junqueira, F., Plachouras, V., Witschel, H.F.: Admission policies for caches of search engine results. In: SPIRE. pp. 74–85 (2007)

[5] Baeza-Yates, R.A., Saint-Jean, F.: A three level search engine index based in query log distribution. In: SPIRE. pp. 56–65 (2003)

[6] Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. ACM Trans. Inf. Syst. 24(1), 51–78 (2006)

[7] Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: WWW. pp. 431–440 (2009)

[8] Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: WWW (2003)

[9] Long, X., Suel, T.: Three-level caching for efficient query processing in large web search engines. In: WWW (2005)

[10] Markatos, E.P.: On caching search engine query results. Computer Communications 24(2), 137–143 (2001), citeseer.ist.psu.edu/markatos00caching.html

[11] Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Johnson, D.: Terrier information retrieval platform. In: ECIR. pp. 517–519 (2005)

[12] Ozcan, R., Altingovde, I.S., Cambazoglu, B.B., Junqueira, F.P., Özgür Ulusoy: A five-level static cache architecture for web search engines. Information Processing & Management (2011), http://www.sciencedirect.com/science/article/pii/S0306457310001081, in press

[13] Ozcan, R., Altingovde, I.S., Ulusoy, O.: Cost-aware strategies for query result caching in web search engines. ACM Trans. Web 5, 9:1–9:25 (May 2011)

[14] Podlipnig, S., Boszormenyi, L.: A survey of web cache replacement strategies. ACM Comput. Surv. 35(4), 374–398 (2003)

[15] Raghavan, V.V., Sever, H.: On the reuse of past optimal queries. In: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 344–350 (1995)

[16] Saraiva, P.C., de Moura, E.S., Ziviani, N., Meira, W., Fonseca, R., Riberio-Neto, B.: Rank-preserving two-level caching for scalable search engines. In: SIGIR (2001)

[17] Skobeltsyn, G., Junqueira, F., Plachouras, V., Baeza-Yates, R.A.: Resin: a combination of results caching and index pruning for high-performance web search engines. In: SIGIR. pp. 131–138 (2008)

[18] Xie, Y., O'Hallaron, D.R.: Locality in search engine queries and its implications for caching. In: INFOCOM (2002)

[19] Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: WWW. pp. 401–410 (2009)

Appendix G

# Paper C.II: Prefetching Query Results and its Impact on Search Engines

Simon Jonassen, B. Barla Cambazoglu and Fabrizio Silvestri.

*Appeared at the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, Portland, OR, USA, August 2012.*

**Abstract:** We investigate the impact of query result prefetching on the efficiency and effectiveness of web search engines. We propose offline and online strategies for selecting and ordering queries whose results are to be prefetched. The offline strategies rely on query log analysis and the queries are selected from the queries issued on the previous day. The online strategies select the queries from the result cache, relying on a machine learning model that estimates the arrival times of queries. We carefully evaluate the proposed prefetching techniques via simulation on a query log obtained from Yahoo! web search. We demonstrate that our strategies are able to improve various performance metrics, including the hit rate, query response time, result freshness, and query degradation rate, relative to a state-of-the-art baseline.

# G.1 Introduction

Commercial web search engines are expected to process user queries under tight response time constraints and be able to operate under heavy query traffic loads. Operating under these conditions requires building a very large infrastructure involving thousands of computers and making continuous investment to maintain this infrastructure [7]. Optimizing the efficiency of the web search systems is important to reduce the infrastructure costs. Even small improvements may immediately translate into significant financial savings for the search engine.

Recent research has shown that result caching [5, 15] is a viable technique to enhance the overall efficiency of search engines. The main idea in result caching is to store the results of frequently or recently processed queries in a large cache and readily serve subsequent occurrences of queries by the cache. This way, significantly more expensive computations at the backend query processing system are avoided, leading to important efficiency gains in the form of reduction in query response latencies and backend query workloads.

Unlike earlier works that have a focus on limited-capacity caches [5, 15], more recent works assume result caches with infinite capacities [11]. The main reason behind this infinite cache assumption is the cheap availability of storage devices that are large enough to store the results of all previous user queries issued to the search engine. Under the infinite cache assumption all future queries can be served by the cache except for compulsory misses which have to be served by the search backend. However, an infinite result cache suffers from the staleness problem [11]. The dynamic nature of web document collections requires frequent index updates, which may render some cache entries stale [1, 8]. In some cases, the search results served by the cache may not be fresh enough in terms of their content and this may degrade the user satisfaction.

In practice, an effective solution to the freshness problem is to associate every result entry in the cache with a time-to-live (TTL) value [2, 11]. In this technique, a cache entry is considered stale once its TTL expires. The hits on the expired entries are treated as misses and are processed by the backend, leading to fresh results. This way, the TTL approach sets an upper bound on the staleness of any search result served by the cache. Unfortunately, it sacrifices some of the efficiency gains achieved by means of result caching. Since the cache hits on the expired cache entries are treated as cache misses, the query traffic volume hitting the backend search system significantly increases with respect to a scenario where no TTL value is associated with the cache entries. In general, the increased backend query volume leads to an increase in the query response latencies and more backend resources are needed to handle the query traffic. Moreover, there is a higher risk that the spikes in the query volume will lead to an overloaded backend, in which case certain queries may have to be processed in the degradation mode, i.e., search results are only partially computed for these queries and the user experience is hampered.

The above-mentioned negative consequences of the TTL approach can be alleviated by combining it with a prefetching[1] strategy that has the goal of updating results of cache

[1]The term prefetching is used in [13, 16] differently to imply requesting the successive results pages for a query.

entries that are expired or about to be expired before a user requests them [11]. An ideal prefetching strategy would have all queries be readily served by the cache. In practice, there is no perfect knowledge of queries that will be issued in the future. Hence, prefetching strategies can only be heuristics.

The observations made before form the main motivation of this paper. In particular, we aim to devise strategies to identify cache entries that are likely to be requested when they are expired. We proactively fetch the associated search results using the idle compute cycles of the backend. The main challenge associated with prefetching is to predict when an expired cache entry will be requested. The predicted times can be used to select queries whose results are worth prefetching and to prioritize them to obtain the highest performance benefit. Although related ideas on cache freshness [11] and batch query processing [12] appear in the literature (see Section G.7), our work is novel in terms of the following contributions.

- We quantify the available opportunity for prefetching (i.e., the amount of backend capacity that can be used for prefetching) and the potential benefit (i.e., the amount of requests for expired cache entries), using a query workload obtained from Yahoo! web search. To best of our knowledge, these were not reported before.
- We propose offline and online strategies to select and prioritize queries that will potentially benefit from prefetching. The offline strategy relies on the observation that the queries tend to repeat on a daily basis and applies query log mining to identify queries whose results are to be prefetched. The online strategy relies on a machine learning model that predicts the next occurrence time of issued queries. Prefetching operations are then prioritized based on these expected times.
- We conduct simulations to observe some important performance metrics, including the hit rate, query response time, freshness, and query degradation rate.

The rest of the paper is organized as follows. In Section G.2, we discuss our system model and motivate the result prefetching problem through observations made over a real-life web search query log. In Sections G.3 and G.4, we present the proposed offline and online prefetching strategies, respectively. We provide the details of our data and experimental setup in Section G.5. The experimental results are presented in Section G.6. We provide the related work in Section G.7. The paper is concluded in Section G.8.

## G.2 Preliminaries

**System architecture.** In this work, we assume the search engine architecture shown in Fig. G.1. User queries are issued to the search engine's main frontend, which contains an infinite result cache where the entries are associated with a TTL value. Queries whose results are found in the cache and not yet expired are readily served by the result cache. Otherwise, they are issued to the frontend of a selected backend search cluster, which is composed of many nodes that host a large index build on the document collection (only one cluster is displayed in the figure). After a query is processed at the backend cluster, the computed results are cached together with the time of the computation so that the expiration time can be determined. The query prefetching module interacts with both the

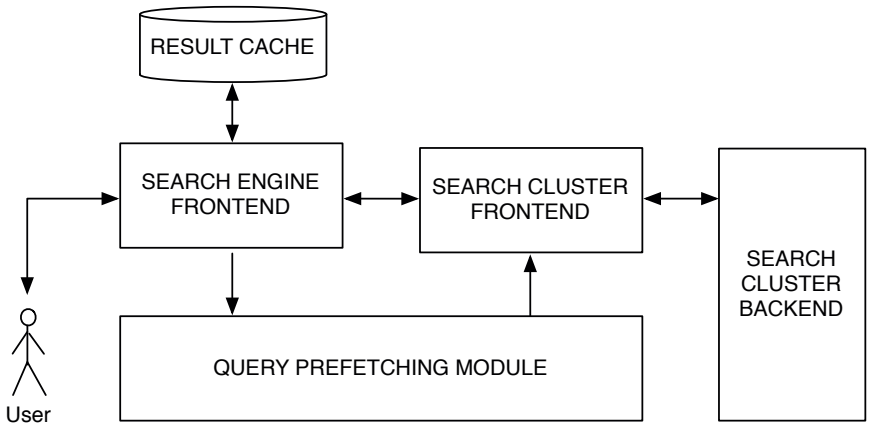



Figure G.1: The sketch of a search engine architecture with a query prefetching module.

search engine frontend and the search cluster frontend. This module is responsible for selecting a set of queries that are expired or about to be expired (from the query logs or the result cache) and issuing them to the search cluster's frontend, which issues them to the backend. The results computed by the backend are then cached like regular user queries.

**Query prefetching problem.** Our focus in this work is on the query prefetching module. The idea behind this module is to avoid, as much as possible, potential cache misses by proactively computing, i.e., prefetching, the results of queries that are about to expire before they are requested by the users. At first glance, the problem of prefetching looks trivial as it seems easy to identify queries that are expired or about to be expired. The problem, however, is quite challenging because not all prefetching operations are useful and prefetching the results of a query consumes backend resources. In particular, if the results of a query are prefetched, but the prefetched results are not requested before they are expired, prefetching leads to waste of resources.

The most important benefit of prefetching is the increase in the cache hit rate [11]. This immediately corresponds to reduced average query response times as more queries can be served by the cache. In addition, the freshness can also be improved if unexpired cache entries are prefetched.[2] Finally, the fraction of queries whose results are computed in the degraded mode can be reduced if prefetching can decrease the amount of processing at peak query traffic times. In summary, the benefits expected from prefetching are reduced query response time, improved result freshness, and reduced query result degradation.

The query results need to be prefetched, as much as possible, when the user query traffic volume is low so that the user queries are not negatively affected from the query processing overhead incurred due to prefetching. Hence, the feasibility of prefetching depends on the availability of the low traffic hours. This raises the question whether the low traffic periods are long enough to prefetch sufficiently many query results. Moreover, the fraction of

[2]It is interesting to note that prefetching only expired cache entries results in increased staleness. In fact, cache staleness not necessarily impacts on results freshness as some expired results might not be retrieved in the future.

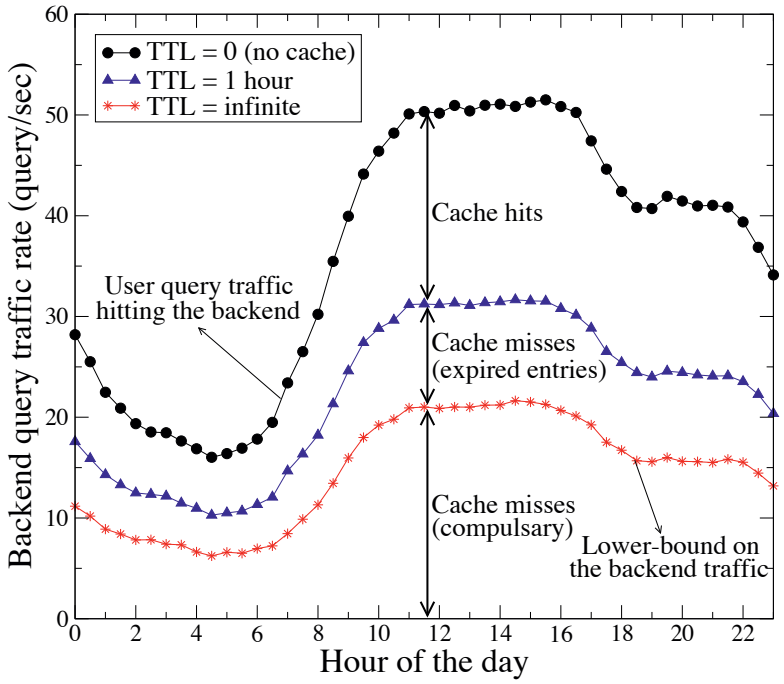



Figure G.2: The user query traffic hitting the backend under different TTL values.

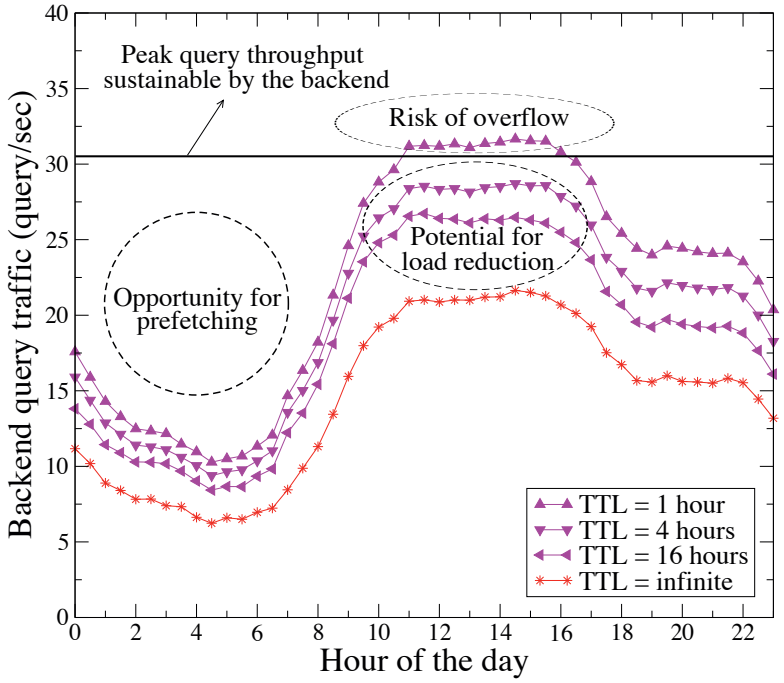



Figure G.3: The variation in the user query traffic distribution within a day and its implications.

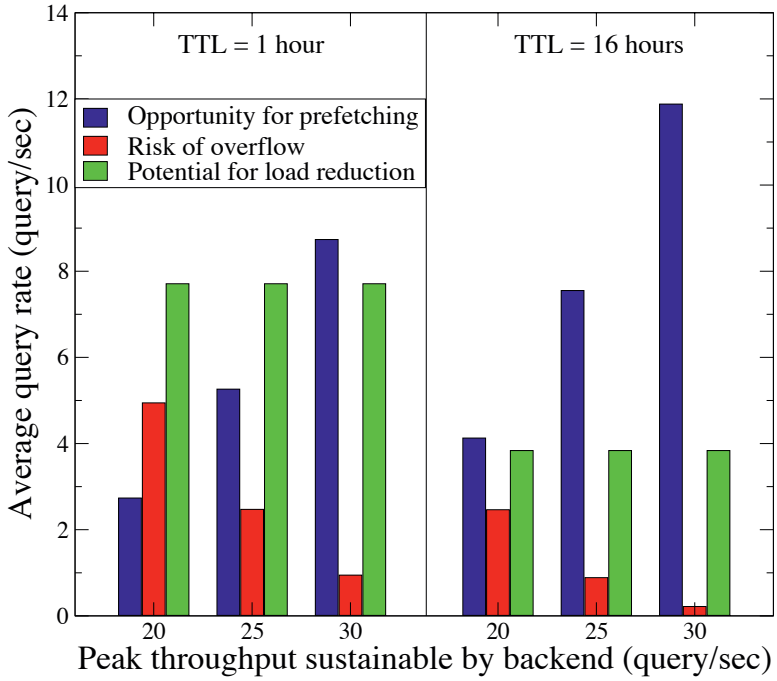



Figure G.4: The opportunity, risk, and potential benefit (quantified as query per second).

queries that can benefit from prefetching needs to be quantified. Finally, the potential risk for query result degradation needs to be identified. We will look into these issues in what follows by analyzing a sample taken from the query traffic received by Yahoo! web search during a particular day.

**Motivating observations.** The upmost curve in Fig. G.2 shows the user query traffic received by the search engine, i.e., the traffic that would hit the backend if there was no result cache. The bottom curve indicates the backend traffic in case of an infinite result cache with an infinite TTL, i.e., only the compulsory cache misses hit the backend. The curve in between shows the backend traffic volume when the TTL is set to one hour. We observe that, during the peak hours, a significant amount of cache misses (about one-third) are due to expired cache entries.

According to Fig. G.3, there is a good amount of opportunity for prefetching (especially during the night) when compared to the amount of requests on expired cache entries. We observe that the traffic volume reaches its peak from 11AM to 4PM, resulting in a risk for query result degradation in scenarios where the backend traffic rate is comparable to the peak sustainable throughput (PST) of the backend (e.g., when the TTL is one hour). In general, the backend query load decreases with increasing TTL values, but the potential for reduction in the query load due to prefetching also decreases since there are fewer expired queries.

In Fig. G.4, we try to quantify the existing opportunity for prefetching, the risk of overflow, and the potential for load reduction, assuming two different TTL values and three different PST values. The reported values are in terms of queries per second, averaged over the

entire day. As an example, with a TTL of one hour and a PST of 20 query/sec, the overflow rate is about five queries per second.

**Prefetching strategies.** We evaluate two alternative types of prefetching strategies. The first set of techniques are offline and rely on query log mining. In these strategies, the prefetched queries are selected from the previous day's query logs. The second set of prefetching strategies are online. The idea here is to predict the next occurrence times of cached queries and try to prefetch expired or about-to-expire cache entries before they are requested while being in an expired state. Both types of strategies have two phases: the selection phase, where the queries whose results are to be prefetched are determined, and the ordering phase, where the selected queries are sorted in decreasing order of their importance and are prefetched in that order.

## G.3 Offline Strategies

There is a high correlation between the repetition of some queries and the time of the day, i.e., some queries are more likely to be issued around the same time of the day. According to Fig. 2 in [10], a large fraction of queries are submitted about 24 hours after their latest submission. Hence, a reasonable strategy is to prefetch certain queries in the previous day's query logs. This strategy is based on query log mining and is completely offline. Hence, the query selection process does not incur any overhead on the runtime system.

### G.3.1 Offline Query Selection

Let $T$ denote the TTL of the result cache. Let $\tau$ denote the current time point and $\tau'$ denote the time point exactly 24 hours before $\tau$. Assume that the time interval $[\tau', \tau)$ is split into $N$ time intervals, each interval having a length of $s = (\tau - \tau')/N$ time units. Let $\langle \delta_1, \delta_2, \ldots, \delta_N \rangle$ be a sequence of $N$ time intervals, where $\delta_i = [\tau' + (i-1) \times s, \tau' + i \times s)$. Each time interval $\delta_i$ is associated with a query bucket $b_i$ that stores the queries issued to the search engine within the respective time interval. The queries to be prefetched during a particular future time interval $[\tau + (i-1) \times s, \tau + i \times s)$ are limited to those in the set of buckets $\langle b_i, b_{i+1}, \ldots, b_{i+\lfloor T/s \rfloor} \rangle$, i.e., the queries issued within the past time interval $[\tau' + (i-1) \times s, \tau' + (i-1) \times s + T)$. Hence, the prefetched queries are selected as

$$\mathcal{Q}_i = \bigcup_{j=i}^{i+\lfloor T/s \rfloor} b_j. \tag{G.1}$$

At the end of a time interval $\delta_i$, we switch to the next query set $\mathcal{Q}_{i+1}$ and start prefetching results of queries in that set.

### G.3.2 Offline Query Ordering

Since it may not be possible to prefetch all queries associated with a time interval, the queries need to be ordered based on their importance, i.e., the benefit that can be achieved

by prefetching a query. While ordering the queries, the past frequency of a query plays an important role. Herein, we discuss three simple strategies to determine the importance of a query.[3] The strategies differ in the way the past query frequency is weighted. In what follows, $w(q)$ represents the weight of a query $q$ and $\mathcal{S}$ is the sample of queries considered in the selection phase mentioned above.

**Unit-weighted frequency.** This strategy assumes that a query's importance is equal to its frequency in $\mathcal{S}$, i.e.,

$$w(q) = \sum_{q_j \in \mathcal{S}} 1, \tag{G.2}$$

where $q_j$ denotes an occurrence of $q$ in the query sample $\mathcal{S}$. This strategy simply prefers prefetching frequent queries as an attempt towards increasing the likelihood of prefetching a query that will repeat again.

**Workload-weighted frequency.** This strategy assigns a heavier weight to those queries appearing during high-traffic periods, i.e., those that are issued when the backend query traffic volume is higher. The motivation here is to prefetch more queries from busy traffic hours so that the number of queries hitting the search cluster backend is reduced in those hours. We compute the weight of a query as

$$w(q) = \sum_{q_j \in \mathcal{S}} v(q_j), \tag{G.3}$$

where $v(q_j)$ represents the backend query traffic volume in queries per second when $q_j$ is observed. This strategy aims at reducing the backend query volume at busy hours and, indirectly, it also aims at reducing the number of degraded query results.

**Time-weighted frequency.** This strategy give a higher priority to queries that occurred closer in time to $\tau'$.

$$w(q) = \sum_{q_j \in \mathcal{S}} (\tau' + T - t(q_j)), \tag{G.4}$$

where $t(q)$ denotes the time at which $q_j$ is issued. The assumption here is that the likelihood of prefetching a repeating query will increase.

## G.4 Online strategies

Online strategies are designed to identify queries that are already expired or about to expire and are going to be issued to the backend due to a TTL miss. Those queries are processed beforehand, possibly when the load at the backend is low. The key feature of this strategy is the prediction of the next time point ($e_q$) at which query $q$ will be issued while its cached results are in an expired state.

[3] It is possible to come up with variants of these strategies.

Table G.1: The features used by the learning model

| Feature type | Feature | Description |
|---|---|---|
| Temporal | `hourOfDay` | Hour of submission |
| | `timeGap` | Time since last occurence |
| Query string | `termCount` | No. of terms in query |
| | `avgTermLength` | Avg. term length |
| Result page | `pageNo` | Requested result page no |
| | `resultCount` | No. of matching results |
| Frequency | `queryFreq` | No. of occ. of query |
| | `sumQueryTF` | Sum |
| | `minQueryTF` | Minimum |
| | `avgQueryTF` | Average |
| | `maxQueryTF` | Max. term query log freq. |
| | `sumDocTF` | Sum |
| | `minDocTF` | Minimum |
| | `avgDocTF` | Average |
| | `maxDocTF` | Max. term document freq. |

In the rest of the section, we use the following notation. We denote by $l_q$ the latest request time and by $u_q$ the latest computation time of the results of query $q$. As in the previous section, $\tau$ denotes the current time and $T$ denotes the TTL. We denote by $s$ the selection period (i.e., we update the prefetching queue at every $s$ time units) and by $f_q$ the number of occurrences of $q$ up to $\tau$.

To estimate $e_q$, each time $q$ is submitted, we predict the time to its next appearance, $n_q$. Under the assumption that $q$ appears every $n_q$ seconds, $e_q$ can be calculated as $l_q+k\times n_q$, where $k$ is the smallest natural number greater than 0 that satisfies $u_q+T<l_q+k\times n_q$, i.e., $k=\lfloor\frac{u_q+T-l_q}{n_q}\rfloor+1$.

The value for $n_q$ is computed through a machine learning model using the features given in Table G.1. Temporal features are based on when a query is submitted. Query string features are based on the syntactic content of the query. Result page features capture the characteristics of the results returned by the search engine. Frequency features are based on counters associated with terms of the queries. We use gradient boosted decision trees to train our model [14].

### G.4.1 Online Selection

The key point in the online strategy is to prefetch queries satisfying $\tau<e_q\leq\tau+T$ when there is processing capacity. As a naïve method of query selection, one could scan the list of queries in the cache and pick those with $e_q$ values that satisfy the constraint. In practice, this is not feasible as the cost of scanning the cache for every submitted query is prohibitive. Hence, we resort to use a bucket data structure where each bucket stores queries with $e_q$ values within a time-period $[t,t+s)$. Insertions and deletions in the bucket list can be realized in $O(1)$-time.

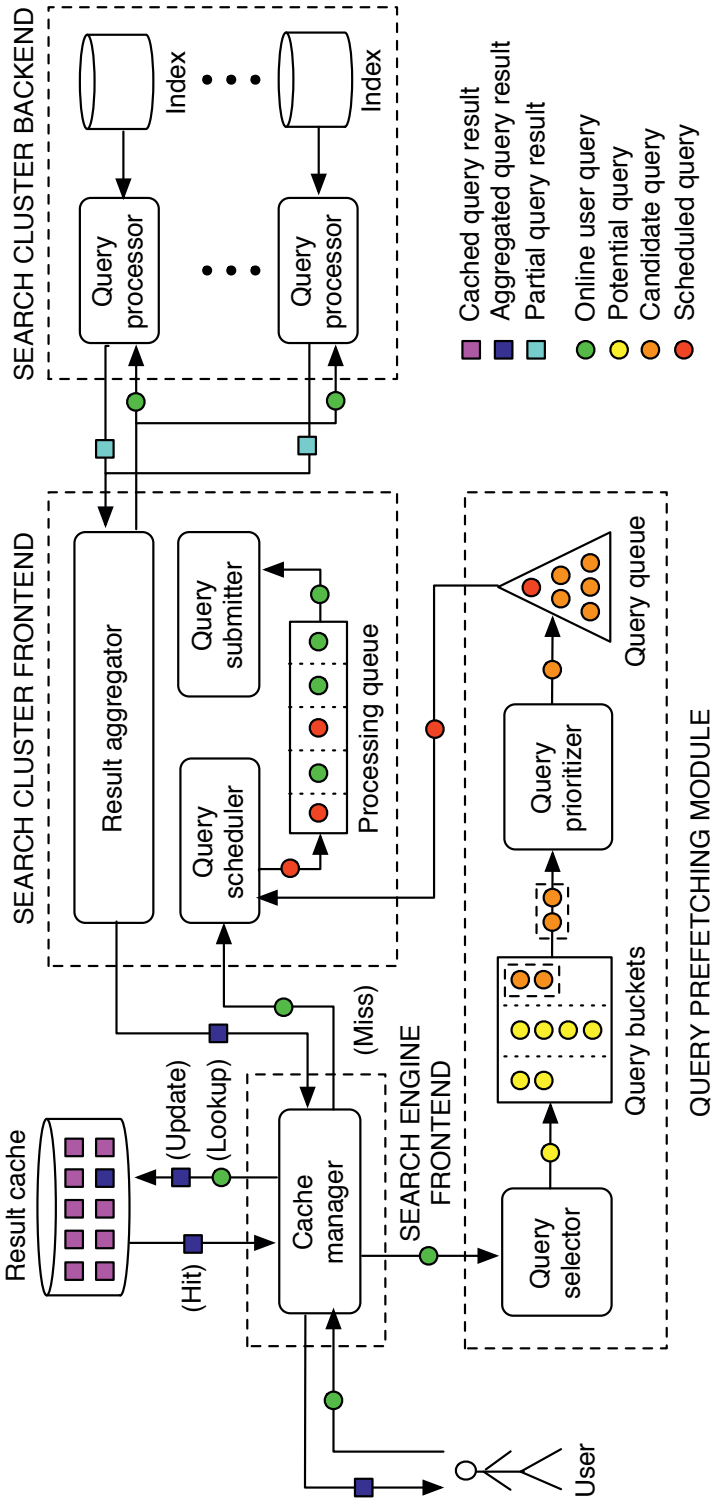



Figure G.5: The proposed search engine architecture with query prefetching capability.

As illustrated in Fig. G.5, the prefetching module maintains a list of buckets. Each time a query is requested, we remove it from the current bucket, modify $e_q$ and place it back in a new bucket. In particular, we have two situations. If the request is a hit, we modify $e_q$ and potentially move the query into another bucket. Otherwise, if it is a miss, we delete it from its current bucket and issue it to the backend. Once it has been processed, we update $e_q$ and eventually insert $q$ in the bucket list. Finally, once in every $s$ seconds we select queries having $e_q \in (\tau, \tau + T]$ from the corresponding buckets and place these in the prefetching queue.

### G.4.2 Online Ordering

To prioritize the queries in the prefetching queue, we evaluate the following three methods:

**Based on age-frequency**. A query is assigned a weight

$$w(q) = (\tau - u_q) \times f_q.$$

In other words, older and more frequent queries get a higher priority. This strategy tends to favor frequent queries that were refreshed long time ago. The goal of this method is to optimize the freshness of the results.

**Based on age-frequency and degradation**. We extend the weight function to include degradation.

$$w(q) = (\tau - u_q) \times f_q \times (1 + d_q).$$

This strategy favors older and degraded queries. The goal of this method is to optimize the freshness of the results and reduce the degradation.

**Based on processing cost and expected load**. Each query is prioritized according to its expected processing cost times the expected difference in the backend load.

$$w(q) = \rho(q) \times L(e_q),$$

where $\rho(q)$ is the estimated processing cost and $L(e_q)$ is the backend load that is observed 24 hours prior to $e_q$. This strategy aims at reducing the query response time and increasing the throughput.

## G.5 Data and Setup

**Prefetching strategies.** We evaluate a number of competing strategies. The first type of strategies are baseline strategies that do not employ any prefetching. These are `no-ttl`, which assumes that there is an infinite result cache with no TTL for entries so that any repetition of a query is a hit, and `no-prefetch`, which assumes that there is an infinite result cache with a TTL, but the prefetching logic is not activated. The second type of strategies are the offline strategies discussed in Section G.3: `off-freq`, `off-vol`, and

`off-time`. The third type of strategies are the online strategies discussed in Section G.4: `on-age`, `on-agedg`, and `on-load`. The fourth type are the oracle versions of online strategies in which the exact next occurrence times of queries are assumed to be known. Following the convention in naming the online strategies, we refer to the oracle strategies as `oracle-age`, `oracle-agedg`, and `oracle-load`. The last implementation is an interpretation of the age-temperature refreshing strategy [11], which we refer to as `age-temp`.

The original `age-temp` strategy maintains a two-dimensional bucket structure, where one of the dimensions corresponds to the result age and the other dimension corresponds to the query frequency (temperature). In our implementation of this strategy, each age bucket is defined by the selection interval $s$ and the temperature is computed as $\log_2(f_q)$, where $f_q$ is the number of times a query is seen since the beginning of the first day. Each time a query is requested, we increase its frequency and potentially move it to a bucket corresponding to a higher temperature. Each time a query is processed, we move it to the bucket with the same temperature but the age dimension corresponding to $\tau$. Furthermore, in our experiments, the temperature of a query cannot decrease. As an alternative, we have evaluated the use of a sliding time-window to dynamically maintain the number of occurrences. However, we have observed that with a window of reasonable size, e.g., 24 or 48 hours, the hit rate tends to be lower. Therefore, in our experiments, we count occurrences starting from the beginning of the first day. Further, we prioritize buckets by the corresponding $\mathrm{age} \times \mathrm{temperature}$ value and restrict selection to the first $30,000$ queries satisfying a minimum age requirement, which prevents too fresh queries from being repeatedly refreshed. We denote the minimum result age as $R$ and use it as the limiting factor for queries selected also by the other methods. Finally, among the selected queries, we give a higher priority to expired queries instead of unexpired ones.

In the `offline` strategies, the only cache state information that is made available to the prefetching module is whether an entry is older than $R$ or not. Other information relies only on the query log from the past day and includes nothing about the current cache state. In the `online` strategies, the decisions are made based on the current cache state and the prediction model mentioned before. As predicting, based on a limited query log, the arrival times of queries that request expired entries is a challenging problem, we allow an entry to be prefetched up to two times without being hit in between.

We use the `oracle` strategies to show the effect of accurate prediction of the next request for an expired entry. The oracle strategies have no information about future hits, neither the number of occurrences nor the time at which they will occur. Nevertheless, they demonstrate that accurate prediction of the next requests for expired entries is enough to achieve good performance. Note that, in our results, we report only the results for the `oracle-agedg` strategy since it always outperforms the other two alternatives.

**Simulation setup.** We sample queries from five consecutive days of Yahoo! web search query logs. The queries in the first three days are used to warmup the result cache. No prefetching is done in these days. At the end of the third day, we start allowing queries into our data structures. The age-temperature baseline starts with queries collected from the first three days. The offline strategies start with queries collected during the third day. The online strategies use the first three days to create a model. As no predictions are made

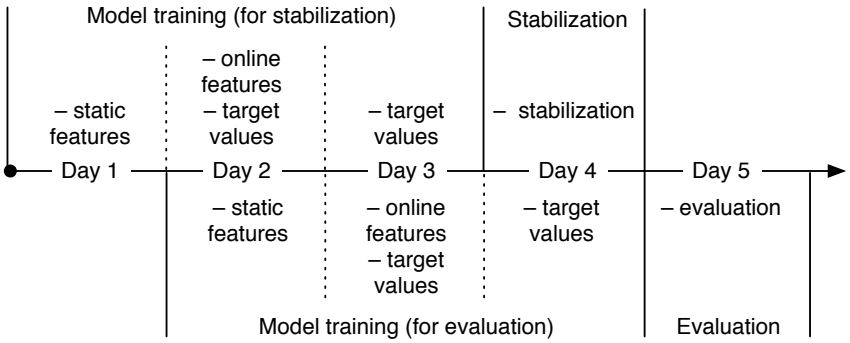



Figure G.6: Temporal splitting of the query log with respect to the tasks.

before the beginning of the fourth day, we use average inter-arrival times to initiate the bucket structure at the end of day three. We assume that the prefetching logic is activated at the beginning of the fourth day. Finally, for all of the techniques, we use the fourth day to stabilize the performance and the fifth day for the performance evaluation.

For the prediction model used by the online strategies, the first day of the training period is used to compute the static features (e.g., query and term frequencies).[4] The instances for which the next arrival times are tried to be predicted are extracted from the second day. The target values (i.e., the next arrival times that are tried to be predicted) are obtained from the second and third days. This setup is illustrated in Fig. G.6. The importance values for the features used by the model are displayed in Fig. G.7. As expected, the query frequency and the time gap between the two consecutive occurrences of a query turn out to be the most important features. For the fourth and fifth days, we train a learning model using the preceding three days.

**System parameters and query processing logic.** The experiments are conducted using a discrete event simulator that takes as input the user queries along with their original submission timestamps. We simulate a system with $K$ compute nodes and $P$ query processors, which allows up to $P$ queries to be concurrently processed with a speedup proportional to $K$. We assume that the entire inverted index is maintained in the memory. In all experiments, we set the TTL ($T$) to 16 hours, which is chosen by an analysis of the opportunity, risk, and potential benefits of prefetching shown in Fig. G.4. We fix $P$ to 8 and vary $K$ to create three different scenarios corresponding to poor ($K = 850$), medium ($K = 1020$), and high performance ($K = 1224$) systems. With 850 nodes, both `no-prefetch` and `no-ttl` experience a degradation of efficiency at peak performance. With 1020 nodes (20% above the first scenario), `no-ttl` reaches the peak sustainable throughput only marginally. With 1224 nodes (20% above the second scenario), both `no-prefetch` and `no-ttl` perform well.

If more than $P$ queries are concurrently issued to the backend, the most recent queries are put into a processing queue. To prevent the queue from growing indefinitely, we monitor

[4]We use a document collection containing several billion documents to compute the features that rely on the document frequencies of terms.

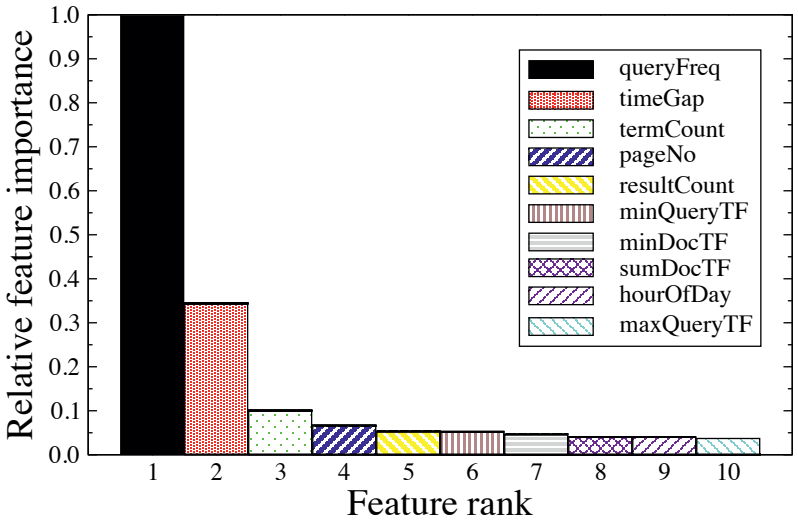



Figure G.7: Feature importance.

the processor usage $P_l$ over the last minute. When $P_l > 0.95 \times P$, we degrade the currently processed query by $P/Q_\tau$, where $Q_\tau$ is the number of queries being processed at the backend at that time.

Since prefetching happens while user queries are scheduled, we monitor the average number of user queries being processed over the last minute $Q_l$ and maintain a budget of $B = \lfloor f \times P - Q_l \rfloor$ processors to be used for prefetching. By default, we use $f = 0.875$ to keep the processor utilization around 87.5 percent. To prevent the system from prefetching recently processed queries, we restrict the minimum age of queries being selected with a parameter $R$. In our experiments we use three different values, TTL/2, TTL/4 and TTL/8. Finally, the prefetching queue is updated/reset once in every $s = 10$ minutes.

**Evaluation metrics.** We evaluate the performance using five different metrics. The metrics are cache hit rate, query response time, average hit age, degradation rate, and backend query traffic rate. The hit rate reflects how good the prefetching algorithm is at having query results available at request. The change in response time reflects the real improvement from prefetching versus its overhead. The upper bound for both metrics is limited by the number of compulsory misses and corresponds to the hit rate and latency of `no-ttl`. The average hit age reflects how good the prefetching method is at keeping cached results fresh. The degradation rate reflects the amount of degraded results being served to the user and is defined as the total degradation divided by the number of queries being answered by the frontend. The total degradation is the sum of the relative degradations of the results returned by the frontend. Finally, the average backend query traffic measures the average number of queries issued to the backend. The lower bound on this number is again defined by `no-ttl`.

## G.6 Experiments

In this section, we present the results of our experiments. Our first observation is that in all the cases that we analyze (except for one), the best possible results are obtained, as expected, by the `oracle-agedg` strategy (the last line in the tables). Obviously, not being feasible in practice, the oracle strategy is presented only as an upper bound to the results that can be attained by the other techniques. Indeed, as the prediction accuracy gets closer to that of the oracle, we should get better and better results.

Note that, in the sole case of the average hit age results reported in Table G.6, the `oracle-agedg` strategy leads to sub-optimal results. This behavior can be explained by observing that the age decreases as we increase the number of times an entry is subject to prefetching. The oracle, instead, minimizes the amount of prefetching and thus some entries (especially those requested further in the future) tend to have higher ages.

Table G.2 shows the cache hit rate of different strategies for various parameters. The `no-prefetch` and `no-ttl` strategies are the two extreme cases. The `no-prefetch` strategy does not involve prefetching and relies only on the TTL to cope with staleness problem. The `no-ttl` strategy correspond to an infinite cache scenario, where only the compulsory misses incur a cost. In between these two extremes, by varying the $K$ and $R$ parameters, we observe that, in all cases, the offline strategies perform worse than the `age-temp` baseline. Our online strategies, however, outperform the baseline. In particular, we compare the hit rate performance of a strategy A versus that of B by computing the relative hit rate improvement with respect to `oracle-agedg` as $I = \frac{\mathsf{HR}_A - \mathsf{HR}_B}{\mathsf{HR}_\mathrm{O} - \mathsf{HR}_B}$, where $HR_A$, $HR_B$, and $HR_\mathrm{O}$ are the hit rates of the A, B, and `oracle-agedg` strategies, respectively. The improvement towards the oracle of our `on-age` versus the baseline `age-temp` ranges from $12\%$ of `on-age` with $K = 1224$ and $R = T/8$ to $22.78\%$ of `on-age` with $K = 850$ and $R = T/4$ with an average of $19.23\%$. Fig. G.8 shows the hit rate trend of four different strategies: `no-prefetch`, `no-ttl`, `on-agedg`, and `oracle-agedg`. According to the figure, our method consistently outperforms the `no-prefetch` strategy.

In Table G.3, we report the response time of our system by varying the simulation parameters. Query response time and hit rate are related and results confirm this behavior. As in the case of hit rate, we report the improvement with respect to the oracle that are similar to those of hit rate figures. The improvement towards the oracle of our `on-age` versus the baseline `age-temp` ranges from $12.4\%$ of `on-age` with $K = 1224$ and $R = T/8$ to $27.5\%$ of `on-age` with $K = 850$ and $R = T/8$ with an average of $21.1\%$. The hourly trend for the average response time is shown in Fig. G.9, where the effect of peak load reduction is evident. From 11AM to 6PM, the peak load response time is greatly reduced. Therefore, in this case, the user experience should be greatly improved.

Another important aspect to consider when evaluating prefetching strategies is the load we put on the backend (Table G.4). If we do not adopt any prefetching strategy, the load on the backend would certainly be affected only by the hit rate. Indeed, the higher the hit rate the less the number of queries hitting the backend. This is confirmed by the numbers in Table G.4 as the `no-prefetch` strategy is the one attaining, in all cases, the

minimum amount of workload at the backend. We remark that our goal is to optimize the exploitation of the infrastructure, i.e., to increase the overall load by keeping the amount of degraded queries as low as possible. Numbers in Table G.4, thus, are more explanatory if read in conjunction with the results presented in Table G.5. Disregarding the case $K = 1224$, in which the computational capacity is too high to be overloaded even by a great number of prefetching operations (with the exception of `on-load` which tries to prefetch expensive queries first), in all cases, `on-age` and `on-agedg` attain the greatest reduction in terms of degraded queries, despite the average backend query traffic is greater than the baseline. Figs. G.10 and G.11 confirm the results reported in the tables and show even more remarkably the effect of prefetching on the backend query traffic, which results to be roughly flattened and far apart from the `no-cache` case. The degradation (Fig. G.11) is also greatly reduced during the peak hours. In this case, we report degradation of query results for the case $K = 850$ instead of $K = 1020$ as in the other cases. This is because, for $K = 1020$, the degradation is negligible.

We also report the average hit age measured in number of minutes passed since the last update. It is worth being pointed out that measuring the average hit ratio does not say much about the quality or staleness of results in the cache. In fact, if the majority of entries are updated but requested after a period of time very close to the TTL, the ideal strategy would not update them. Indeed, this would make the average hit age increase when those queries are actually requested only slightly before their expiration. We report these results in Table G.6 and Fig. G.12.

Finally, we measure the accuracy of prefetching methods. We define the accuracy as the fraction of correct prefetching actions. That is, the average between the fraction of prefetching operations never followed by a hit with respect to the total number of prefetching operations and the fraction of compulsory misses relative to the number of misses. As illustrated in Table G.7, while performing less prefetching, the offline techniques have in general higher accuracy. In addition, it is worth to note that there is a high potential for improvement. If we compare prefetching accuracy with that of `oracle-agedg`, we can observe that we are still far from being close to the maximum. On the other hand, the worst accuracy is achieved by `on-load`, which illustrates that prioritizing frequent queries maximizes the probability of useful prefetching. We note that the `age-temp` baseline performs poorly with respect to our strategies.

In what follows, we enumerate what we retain to be the take away messages from this work. First, the baseline relies only on the knowledge of the past frequency and the current age. This approach cares about keeping popular entries fresh and does not care if they will be used again in the future. Second, the offline strategies rely only on the information about the queries submitted in the previous day. Even though they are worse than the baseline, they perform surprisingly well, given that they use only one day of history. Third, the online strategies rely on predicted future query expirations. This prediction task is very difficult to carry out, as experiments show, yet a mildly good prediction policy gives very good results in terms of search backend exploitation, low query degradation, and hit-ratio. Fourth, even if oracle has a perfect knowledge of all the future misses, it does not have all the information needed to order the queries in the best possible way to reduce the number of misses, for instance. However, we use the oracle strategies as an upper bound to the

Table G.2: Cache hit rate

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| `no-prefetch` | 0.493 | – | – | 0.494 | – | – | 0.494 | – | – |
| `no-ttl` | 0.600 | – | – | 0.600 | – | – | 0.600 | – | – |
| `age-temp` | 0.517 | 0.517 | 0.516 | 0.533 | 0.533 | 0.533 | 0.542 | 0.542 | 0.543 |
| `off-freq` | 0.512 | 0.512 | 0.510 | 0.515 | 0.515 | 0.514 | 0.517 | 0.518 | 0.517 |
| `off-vol` | 0.511 | 0.511 | 0.509 | 0.514 | 0.514 | 0.513 | 0.516 | 0.517 | 0.517 |
| `off-time` | 0.511 | 0.511 | 0.509 | 0.515 | 0.515 | 0.515 | 0.519 | 0.521 | 0.521 |
| `on-age` | 0.534 | 0.535 | 0.533 | 0.546 | 0.547 | 0.547 | 0.549 | 0.550 | 0.553 |
| `on-agedg` | 0.534 | 0.535 | 0.533 | 0.546 | 0.547 | 0.547 | 0.549 | 0.550 | 0.553 |
| `on-load` | 0.493 | 0.493 | 0.494 | 0.494 | 0.494 | 0.495 | 0.494 | 0.494 | 0.497 |
| `oracle-agedg` | 0.597 | 0.596 | 0.592 | 0.599 | 0.599 | 0.600 | 0.600 | 0.600 | 0.600 |

Table G.3: Average query response time (in ms)

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| `no-pefetch` | 309.1 | – | – | 157.5 | – | – | 112.5 | – | – |
| `no-ttl` | 183.6 | – | – | 122.0 | – | – | 95.5 | – | – |
| `age-temp` | 278.1 | 277.4 | 277.4 | 153.8 | 153.7 | 153.9 | 119.0 | 119.0 | 119.0 |
| `off-freq` | 280.5 | 279.9 | 280.1 | 157.2 | 156.2 | 155.3 | 120.3 | 117.8 | 114.8 |
| `off-vol` | 280.1 | 280.6 | 282.3 | 157.6 | 156.9 | 155.9 | 120.6 | 118.6 | 116.3 |
| `off-time` | 280.9 | 283.0 | 285.5 | 157.8 | 157.1 | 156.9 | 120.6 | 118.8 | 117.2 |
| `on-age` | 257.8 | 255.5 | 257.5 | 148.1 | 147.9 | 147.9 | 116.6 | 116.5 | 116.4 |
| `on-agedg` | 253.8 | 254.8 | 257.3 | 147.9 | 147.7 | 147.7 | 116.6 | 116.5 | 116.4 |
| `on-load` | 316.2 | 311.2 | 312.4 | 173.3 | 173.2 | 171.2 | 133.7 | 133.3 | 132.5 |
| `oracle-agedg` | 189.7 | 189.8 | 191.2 | 129.5 | 129.0 | 128.9 | 99.7 | 99.4 | 99.0 |

Table G.4: Average backend query traffic rate (query/sec)

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| `no-prefetch` | 18.34 | – | – | 18.34 | – | – | 18.34 | – | – |
| `no-ttl` | 14.50 | – | – | 14.50 | – | – | 14.50 | – | – |
| `age-temp` | 23.48 | 23.44 | 23.41 | 27.28 | 27.09 | 26.93 | 32.75 | 32.44 | 32.07 |
| `off-freq` | 23.01 | 21.00 | 19.97 | 25.22 | 22.62 | 20.74 | 29.03 | 24.67 | 21.65 |
| `off-vol` | 22.98 | 20.97 | 19.95 | 25.19 | 22.59 | 20.83 | 29.00 | 24.97 | 22.23 |
| `off-time` | 23.04 | 21.18 | 20.08 | 25.15 | 22.73 | 21.28 | 29.23 | 25.50 | 22.92 |
| `on-age` | 25.11 | 24.93 | 24.33 | 30.34 | 30.03 | 29.15 | 36.99 | 36.43 | 35.26 |
| `on-agedg` | 25.07 | 24.86 | 24.28 | 30.35 | 30.03 | 29.15 | 36.99 | 36.43 | 35.26 |
| `on-load` | 18.95 | 19.03 | 19.24 | 19.53 | 19.60 | 19.88 | 20.78 | 20.86 | 21.17 |
| `oracle-agedg` | 20.92 | 20.49 | 19.85 | 22.47 | 21.63 | 20.74 | 23.99 | 23.19 | 21.62 |

Table G.5: Average degradation rate ($\times 10^{-4}$)

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| `no-prefetch` | 166.9 | – | – | 1.398 | – | – | 0.000 | – | – |
| `no-ttl` | 7.2 | – | – | 0.003 | – | – | 0.000 | – | – |
| `age-temp` | 76.6 | 79.0 | 95.0 | 0.485 | 0.412 | 0.418 | 0.000 | 0.000 | 0.000 |
| `off-freq` | 87.9 | 88.2 | 89.8 | 0.645 | 0.731 | 0.849 | 0.000 | 0.000 | 0.000 |
| `off-vol` | 88.8 | 87.8 | 87.9 | 0.632 | 0.606 | 0.645 | 0.000 | 0.000 | 0.000 |
| `off-time` | 94.0 | 91.1 | 93.4 | 0.807 | 0.661 | 0.689 | 0.000 | 0.000 | 0.000 |
| `on-age` | 50.3 | 48.8 | 51.3 | 0.319 | 0.223 | 0.118 | 0.000 | 0.000 | 0.000 |
| `on-agedg` | 50.6 | 49.5 | 49.6 | 0.201 | 0.169 | 0.284 | 0.000 | 0.000 | 0.000 |
| `on-load` | 233.2 | 241.5 | 175.4 | 173.3 | 173.2 | 171.3 | 133.7 | 133.3 | 132.5 |
| `oracle-agedg` | 5.7 | 5.7 | 6.4 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

Table G.6: Average hit age (in minutes)

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| no-prefetch | 357 | – | – | 356 | – | – | 356 | – | – |
| no-ttl | 4737 | – | – | 4735 | – | – | 4734 | – | – |
| age-temp | 342 | 356 | 375 | 170 | 209 | 289 | 137 | 170 | 246 |
| off-freq | 222 | 255 | 322 | 139 | 172 | 239 | 111 | 147 | 227 |
| off-vol | 222 | 254 | 327 | 138 | 172 | 329 | 111 | 148 | 226 |
| off-time | 226 | 259 | 329 | 144 | 179 | 242 | 118 | 155 | 232 |
| on-age | 217 | 255 | 333 | 135 | 163 | 235 | 114 | 148 | 222 |
| on-agedg | 216 | 254 | 333 | 135 | 163 | 235 | 114 | 148 | 222 |
| on-load | 356 | 356 | 353 | 356 | 356 | 355 | 352 | 352 | 360 |
| oracle-agedg | 272 | 291 | 336 | 258 | 280 | 324 | 265 | 287 | 332 |

Table G.7: Prefetching accuracy

| | $K=850$ | | | $K=1020$ | | | $K=1224$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Strategy | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ | $R=T/8$ | $R=T/4$ | $R=T/2$ |
| age-temp | 0.529 | 0.525 | 0.514 | 0.550 | 0.543 | 0.533 | 0.543 | 0.535 | 0.523 |
| off-freq | 0.543 | 0.593 | 0.629 | 0.551 | 0.586 | 0.610 | 0.534 | 0.563 | 0.592 |
| off-vol | 0.537 | 0.588 | 0.622 | 0.546 | 0.580 | 0.602 | 0.531 | 0.556 | 0.573 |
| off-time | 0.534 | 0.573 | 0.601 | 0.542 | 0.572 | 0.586 | 0.533 | 0.554 | 0.561 |
| on-age | 0.599 | 0.592 | 0.594 | 0.597 | 0.587 | 0.588 | 0.570 | 0.560 | 0.557 |
| on-agedg | 0.599 | 0.593 | 0.595 | 0.597 | 0.587 | 0.588 | 0.570 | 0.560 | 0.557 |
| on-load | 0.412 | 0.441 | 0.528 | 0.417 | 0.451 | 0.516 | 0.419 | 0.475 | 0.495 |
| oracle-agedg | 0.842 | 0.846 | 0.855 | 0.798 | 0.799 | 0.804 | 0.723 | 0.720 | 0.726 |

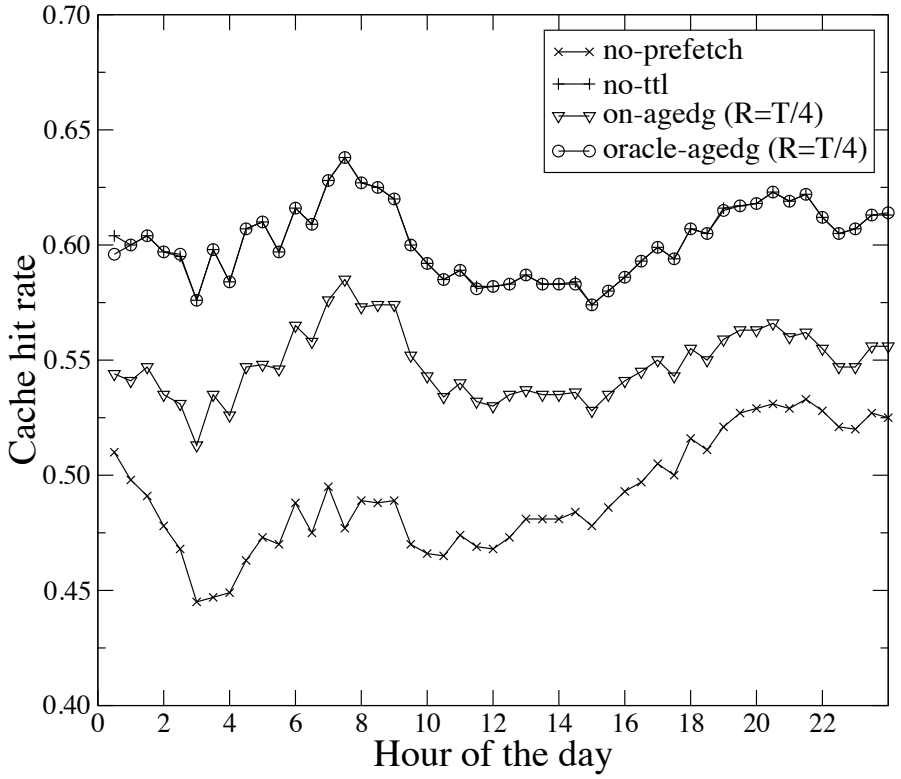



Figure G.8: Result cache hit rate ($K = 1020$).

effectiveness of prefetching with respect to the cache content. Finally, in Figs. G.8–G.12, we observe that, during the peak hours, the benefits of prefetching are greatly amplified thus confirming our initial hypothesis, i.e., prefetching has a great potential for improving search engine utilization and peak-load performance.

## G.7 Previous Work

Herein, we survey the previous work on result caching, freshness, and batch query processing. Interested readers may refer to [10] for a broader survey on search efficiency.

**Result caching.** So far, a large body of research focused on increasing the result cache hit rates [4, 20] or to reduce the query processing cost of backend search systems [15, 21]. Depending on how the cache entries are selected, static [6], dynamic [22], or hybrid [13] caching strategies are followed. The earlier works assumed limited-capacity caches, where the main research issues are admission [5], prefetching [16], and eviction [20], whereas the recent works mostly adopted an infinite cache assumption [11]. A number of proposals are made to combine other layers of caching with result caching, resulting in two-level [3, 22], three-level [17, 18], or even five-level [19] caching architectures.

**Cache freshness.** The most recent works deal with result caching in the context of maintaining the freshness of the results served by the cache [1, 8, 9, 11]. In this line of research,

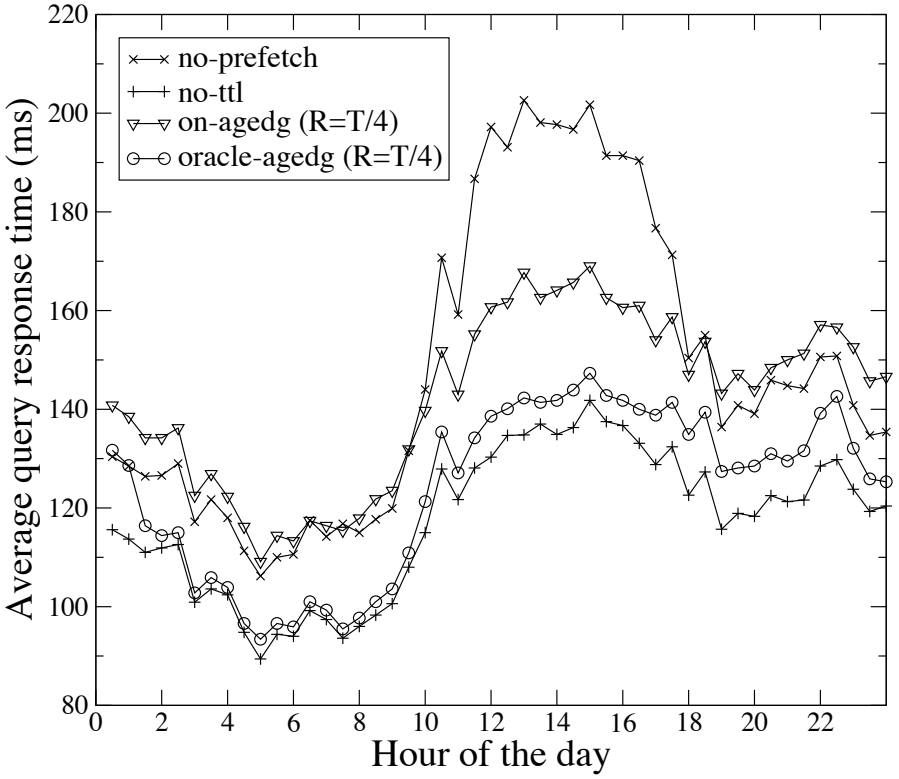



Figure G.9: Average query response time ($K = 1020$).

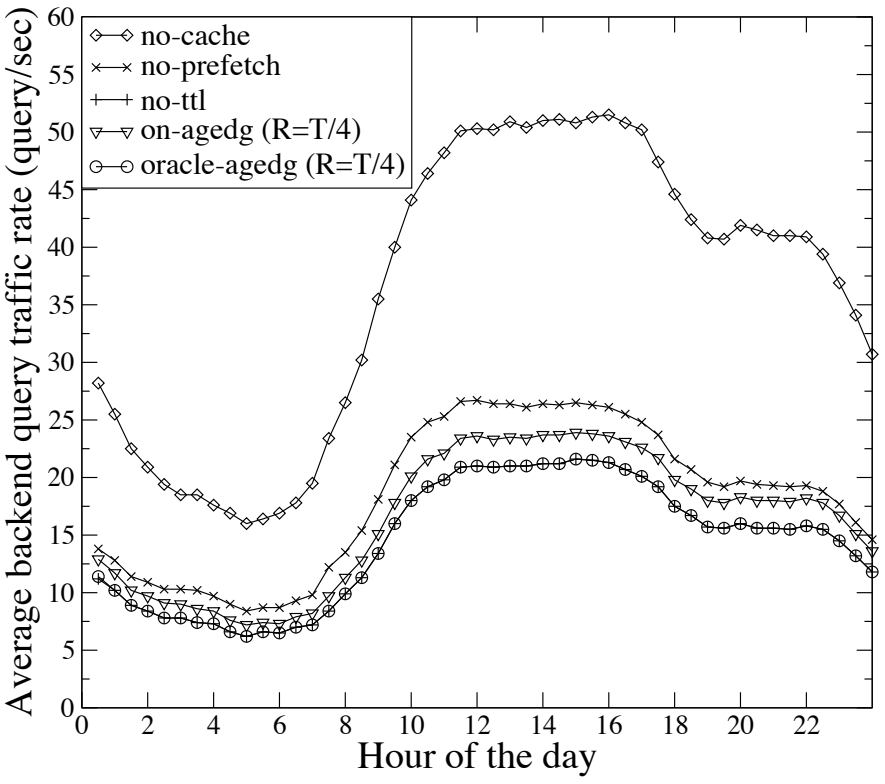



Figure G.10: Backend query traffic rate (only user queries, $K = 1020$).

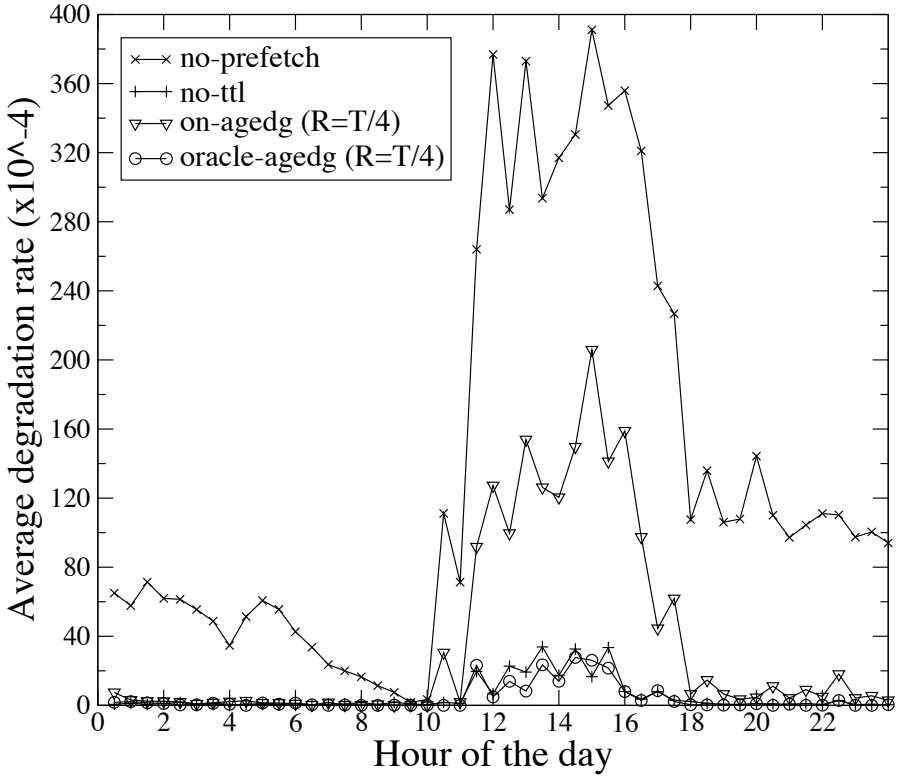



Figure G.11: Average degradation rate ($K = 850$).

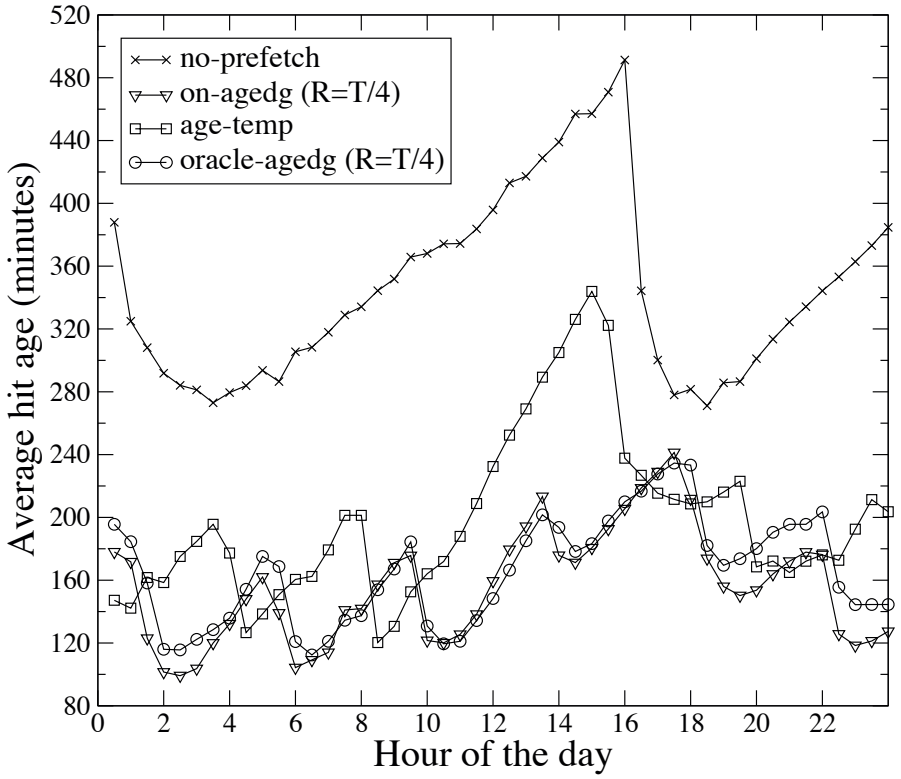



Figure G.12: Average hit age ($K = 1020$).

the cache entries are associated with TTL values that are fixed for all queries. In [2], each query is associated with a different TTL value depending on its freshness requirements. In several works, the TTL approach is coupled either with more sophisticated techniques such as invalidation of potentially stale cache entries [1, 8, 9], where the goal is to predict the stale cache entries by exploiting certain information obtained during index updates. The search results whose freshness is potentially affected by the updates are then marked as invalid so that they are served by the backend system in a successive request, rather than by the cache. The approach followed in [11], on the other hand, relies on proactive refreshing of cached search results. In this approach, the indexing system does not provide any feedback on staleness. Instead, some presumably stale search results are selected and refreshed based on their likelihood of being requested in future and also depending on the availability of free processing cycles at the backend search system. Our work proposes strategies that are alternative to that in [11] and also focuses on different performance metrics.

**Batch query processing.** In [12], some efficiency optimizations are proposed for batch query processing in web search engines. Those optimizations (e.g., query reordering) are orthogonal to ours and, in case of a partially disk-resident index, they may be coupled with our prefetching techniques. Since we assume an in-memory index, however, we did not consider those optimizations in our work.

## G.8 Conclusions and Further Work

We investigated the impact of prefetching on search engines. We showed that an oracle strategy that has a perfect information of future query occurrences can achieve the best attainable results. We made an attempt to close the gap with the oracle by testing two alternative set of prefetching techniques: offline and online prefetching. We showed that the key aspect in prefetching is the prediction methodology. Furthermore, we showed that our accuracy in predicting future query occurrences is only about a half of the best that can be done. We plan to extend this work in several directions. First, we are going to design more effective techniques for predicting the expiration of a query. Second, we are going to evaluate the economic impact of prefetching on the search operations. Finally, we would like to design speculative prediction techniques that will be able to prefetch queries that were not even submitted. This would reduce the number of compulsory misses that form an upper bound on the cache performance.

## G.9 Acknowledgments

Simon Jonassen was an intern at Yahoo! Research and supported by the iAd Centre, Research Council of Norway and NTNU.

## G.10 References

[1] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Timestamp-based result cache invalidation for web search engines. In *Proc. 34th Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 973–982, 2011.

[2] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In *Proc. 34th European Conf. Information Retrieval*, pages 401–412, 2012.

[3] I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Second chance: a hybrid approach for dynamic result caching in search engines. In *Proc. 33rd European Conf. Information Retrieval*, pages 510–516, 2011.

[4] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. 30th Annual Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 183–190, 2007.

[5] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. Witschel. Admission policies for caches of search engine results. In N. Ziviani and R. Baeza-Yates, editors, *String Processing and Information Retrieval*, volume 4726 of *Lecture Notes in Computer Science*, pages 74–85. Springer Berlin / Heidelberg, 2007.

[6] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *String Processing and Information Retrieval*, volume 2857 of *Lecture Notes in Computer Science*, pages 56–65. Springer Berlin / Heidelberg, 2003.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: the Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *Proc. 33rd Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 82–89, 2010.

[9] E. Bortnikov, R. Lempel, and K. Vornovitsky. Caching for realtime search. In *Proc. 33rd European Conf. Advances in Information Retrieval*, pages 104–116, 2011.

[10] B. B. Cambazoglu and R. Baeza-Yates. Scalability challenges in web search engines. In M. Melucci, R. Baeza-Yates, and W. B. Croft, editors, *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series*, pages 27–50. Springer Berlin Heidelberg, 2011.

[11] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *Proc. 19th Int'l Conf. World Wide Web*, pages 181–190, 2010.

[12] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proc. 4th ACM Int'l Conf. Web Search and Data Mining*, pages 137–146, 2011.

[13] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Information Systems*, 24(1):51–78, 2006.

[14] J. H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38:367–378, February 2002.

[15] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proc. 18th Int'l Conf. World Wide Web*, pages 431–440, 2009.

[16] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. 12th Int'l Conf. World Wide Web*, pages 19–28, 2003.

[17] H. Li, W.-C. Lee, A. Sivasubramaniam, and C. L. Giles. A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proc. 7th Int'l Conf. Web Engineering*, pages 121–136, 2007.

[18] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. 14th Int'l Conf. World Wide Web*, pages 257–266, 2005.

[19] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *Proc. 19th ACM Int'l Symp. High Performance Distributed Computing*, pages 215–226, 2010.

[20] E. P. Markatos. On caching search engine query results. *Computer Comm.*, 24(2):137–143, 2001.

[21] R. Ozcan, I. S. Altingövde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):9:1–9:25, 2011.

[22] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. 24th Annual Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, pages 51–58, 2001.

# Additional Material

*"If computers get too powerful, we can organize them into a committee – that will do them in."*

– Bradley's Bromide

# Appendix H

# A Retrospective Look at Partitioned Query Processing

In this appendix we present some of the unpublished results and discuss how our improvements of pipelined query processing influence the comparison between term-wise and document-wise partitioning.

In 2008, we performed a simulation-based evaluation of several partitioning methods. Figure H.1 summarizes the main results of my master thesis [81, 83] obtained with help of TREC GOV2, and a small subset of TREC Terabyte Track 2005 Efficiency Topics (E05) corresponding to 50 000 ms of execution time. A comprehensive description of the framework, algorithms and parameters can be found in the master thesis [81]. In brief, with document-wise partitioning (DP) each node processes a query replica and returns the $k$ best results to the query broker, which further selects the $k$ best results. With term-wise partitioning (TP), each node partially processes its sub-query and transfers the partial results to the query broker, which further combines them and selects the $k$ best results. With pipelined query processing (PL), each node in the route retrieves any actual posting lists, combines them with the received accumulators and sends them to the next node. The last node in the route extracts the $k$ best results. Finally, neither of the methods apply any skipping or pruning optimizations and the system simulated 8 processing nodes.

As Figure H.1 shows, for both disjunctive (OR) and conjunctive (AND) queries, DP clearly outperforms pure TP, while PL outperforms DP in terms of throughput at higher multiprogramming levels. Moreover, the benefit of applying PL in combination with high multiprogramming levels is more significant for AND queries. On the other hand, DP provides a superior performance at lower multiprogramming levels.

A better interpretation of this results is as follows. Under conditions when long query latency can or have to be tolerated, such as in case of offline query processing or when the system is heavy loaded, PL may provide higher throughput and therefore is more advantageous. However, under normal conditions or when query latency has to be constrained, DP is the method of choice.

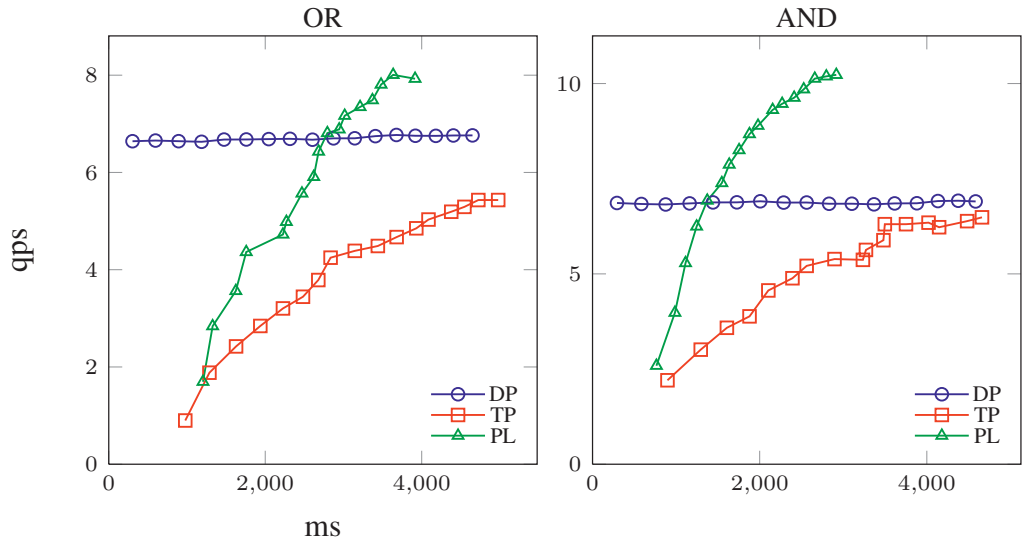
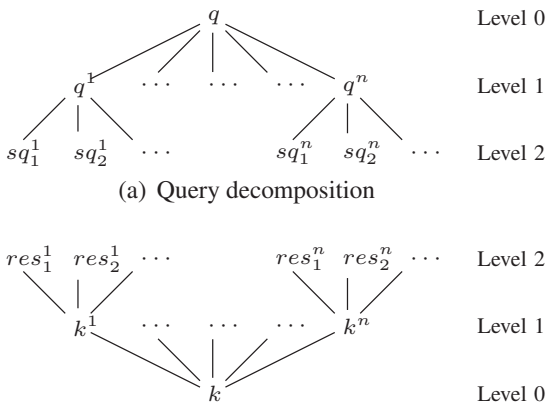



Figure H.1: Throughput and latency with varied multiprogramming level (2,4,6,. . . ,34) obtained from simulation [81, 83].

As an attempt to find a trade-off between term-wise and document-wise partitioning, in 2010, we implemented and evaluated 2D partitioning using Dogsled (see Sec. 3.5.1). In this work, the nodes were logically arranged in a $n \times m$ grid corresponding to $n$ document and $m$ term partitions. Both the index (GOV2) and the query processing system were equivalent to the one used in Paper A.II, and the experiments were based on 5 000 warm-up and 5 000 test queries obtained from E05.




(a) Query decomposition

(b) Combination of results

Figure H.2: Query decomposition and processing with a 2D partitioned index.

As we illustrate in Figure H.2, the query processing model used for 2D partitioning is organized in three logical levels. At level 0, a node is chosen as the receptionist for a particular query in a round-robin manner. This node broadcasts the the query to one of the nodes in each of the document partitions. At level 1, each of these nodes divides the query into several sub-queries and sends them to the corresponding nodes within the same

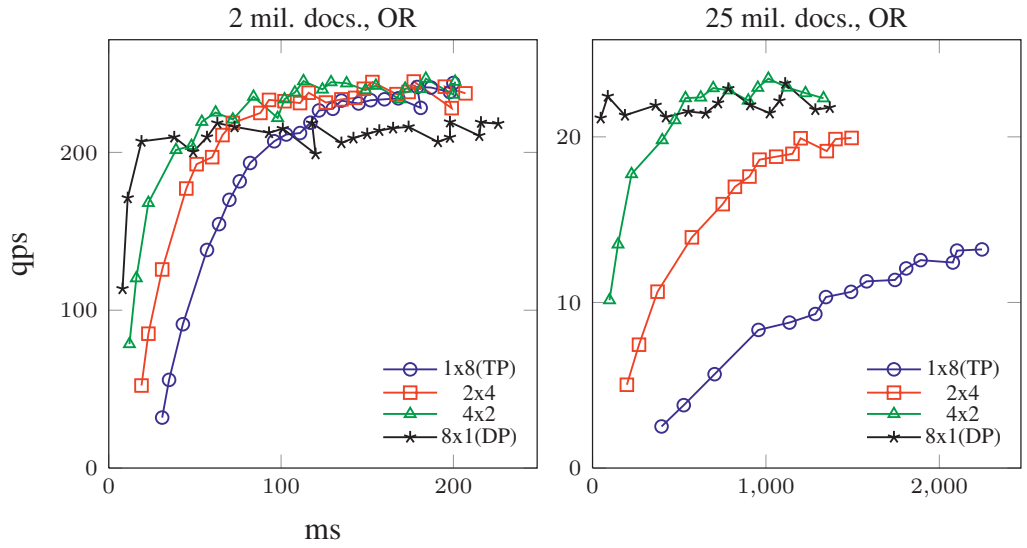



Figure H.3: Throughput and latency with varied multiprogramming level (1,2,4,6,...) obtained from the experiments with Dogsled.

document partition. At level 2, each of these nodes retrieves and processes the postings and returns the partially scored candidates to the corresponding node at the level above. The nodes at level 1 combine the partial results, select the $k$ best results and return them to the receptionist node. The receptionist finally selects the $k$ best results and returns the to the broker. Moreover, if $n=1$ or $m=1$, query processing involves only two levels.

Figure H.3 illustrates the performance of 2D partitioning with two indexes corresponding to a 2 mil. document subset and a complete GOV2 index. These results show that with the smaller index, TP provides superior throughput at higher multiprogramming levels, while DP is more efficient at lower multiprogramming levels. In this case, 2D partitioning allows to trade-off between the two methods and to achieve higher throughput at intermediate multiprogramming levels. However, for a complete GOV2 index, the results are more pessimistic. Because TP is clearly less efficient than DP, 2D partitioning is barely able to reach the throughput level of DP.

These results motivated us to look at pipelined query processing, and consequently to come up with the ideas described in Papers A.II, A.III and A.IV. The main intention was either to eliminate the gap between the two methods, or at least to reduce it so that 2D partitioning would be more beneficial. Therefore, an important question is whether our contributions and efforts have actually improved the performance of TP compared to DP.

In order to answer this question, in 2012, we extended Laika (see Sec. 3.5.1) with query processing over a document-wise partitioned index. We summarize the results in Figure H.4. Here, PL+OR and PL+AND represent the $\text{MSD}^{\star}_{\text{S}}$ and $\text{AND}_{\text{S}}$ methods from Paper A.III. Consequently, DP+AND combines document-wise partitioning and AND query processing with skipping, while DP+OR applies Max-Score query processing and skipping. The query logs used in these experiments correspond to what described in Paper A.III, subsets of E05 and E06 with 5 000 warm-up and 15 000 test queries.

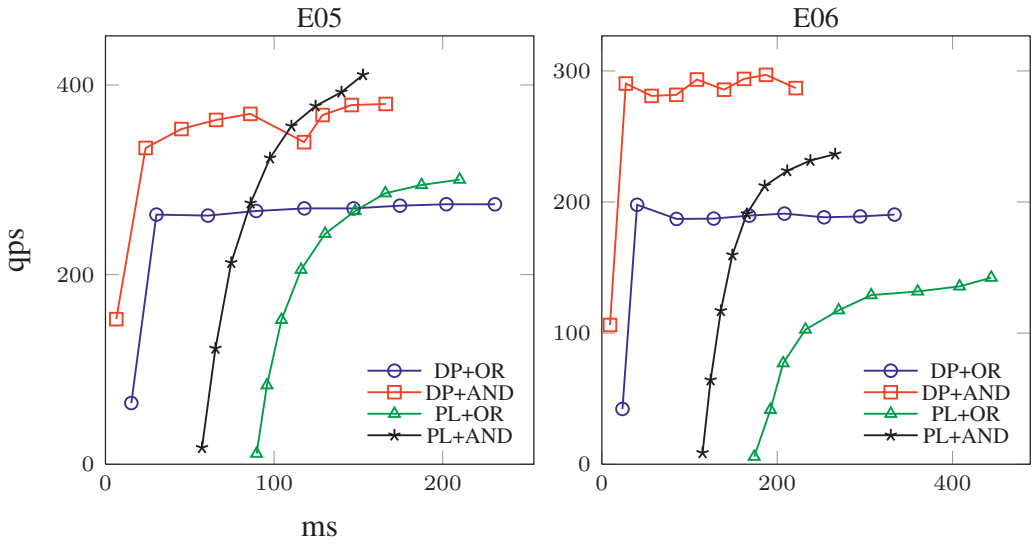



Figure H.4: Throughput and latency with varied multiprogramming level (1,8,16,. . . ,64) obtained from the experiments with Laika.

As described in Paper A.III, E05 contains queries that are less correlated to the document collection, while E06 contains highly correlated queries that in combination with stemming overload the system. As the results show, with E05, PL outperforms DP in terms of maximum throughput at higher multiprogramming levels for both AND and OR queries. In this case, the techniques described in Paper A.IV can be used to further reduce the gap between the two methods at intermediate and lower multiprogramming levels. However, with E06, the results are less optimistic and show that DP is a clear winner.

Of course, Paper A.III suggests several extensions that may further improve the performance of PL. However, even if we succeed to improve PL so much that it may outperform DP on one system, this may not apply on another system.

The main problem lies in the nature of TP and scalability. Skipping and pruning have been shown to improve PL, but they also improve DP. While they may provide a significant improvement on a fixed system, latency will still grow with increasing collection size. For PL, latency is constrained by the longest posting list (i.e., the number of documents in the collection) in terms of I/O, computation or communication cost. The techniques described in Paper A.IV can improve query latency by utilizing available cores, but assuming that the number of cores is fixed, the speed-up will always be limited. Additionally, load imbalance and communication overhead induced by TP are hard to deal with. On the other hand, for DP, latency can be scaled by adding more nodes. With respect to throughput, DP can be combined with replication and a growing system can be scaled in both dimensions [128, 130]. While a proper evaluation would be required, we doubt that our techniques or TP in general can compete with a combination of DP and replication in terms of scalability.

Our conclusion is therefore twofold. On one hand, we have contributed with several novel techniques, which under careful evaluation have shown a significant improvement over the state-of-the-art TP/PL baseline. The collection size and the number of nodes used in our experiments are comparable to what is common among the related work published in

top IR venues. On the other hand, we admit that despite all our efforts DP remains more advantageous and that TP may not be able to outperform DP.