Karlsruhe Institute of Technology

Faster Full Text Search through Document Clustering

Diploma Thesis

Jonathan Dimond

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer/Advisor: Prof. Dr. rer. nat. Peter Sanders

January 1, 2013 – July 31, 2013

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, July 31, 2013

. . . .

(Jonathan Dimond)

Faster Full Text Search through Document Clustering

Jonathan Dimond <mail@dimond.de>

Typeset using LaTeX on July 31, 2013, 09:21

Zusammenfassung

Schneller und einfacher Zugang zu Informationen ist einer der Grundpfeiler unserer heutigen Informationsgesellschaft. Eine wichtige Technik im Bereich von *Information Retrieval* stellt die Volltextsuche dar. Volltextsuche ist nach wie vor ein Gebiet der aktiven Forschung. Eine schnellere Volltextsuche führt zu höherem Durchsatz, geringeren Hardwarekosten und einer verbesserten Benutzerfreundlichkeit durch geringere Latenzen.

Volltextsuchen benutzen in der Praxis häufig einen *invertierten Index*. Ein invertierter Index speichert für jedes Wort die Menge an Dokumenten die dieses Wort enthalten. Eine häufige Form von Suchanfragen sind konjunktive Suchanfragen bei denen Dokumente gesucht werden, die alle Wörter aus der Suchanfrage enthalten. Konjunktive Suchanfragen können mit einem invertiertem Index beantwortet werden, indem die Mengen der Dokumente für alle Wörter aus der Suchanfrage geschnitten werden.

Diese Diplomarbeit beschäftigt sich mit der Frage, wie die Einteilung von Dokumenten in Cluster die Berechnung dieser Schnitte beschleunigen kann. Ein Cluster bezeichnet in dieser Arbeit eine Menge von zusammengehörigen Dokumenten. Die Idee ist einfach, aber neu. Moderne und effiziente Schnittalgorithmen haben eine Zeitkomplexität, die von der Größe der kleinsten zu schneidenden Menge dominiert wird. Wenn Dokumente so in Cluster aufgeteilt werden, dass Wörter möglichst ungleichmäßig auf die Cluster verteilt werden entsteht ein positiver Effekt für die Schnittalgorithmen. In den meisten Clustern kommt jeweils ein Wort der Suchanfrage nur sehr selten vor und die Schnittalgorithmen profitieren von einer kürzeren Laufzeit durch eine geringere Größe der kleinsten Menge. Über alle Cluster summiert ergibt sich so eine deutlich geringere Zeitkomplexität.

Die Basis der Diplomarbeit bildet eine neuartige und effizient berechenbare Bewertungsfunktion die diese Idee umsetzt. Die Bewertungsfunktion optimiert gezielt die erwartbare Zeitkomplexität von Schnitten für Suchanfragen bei der Zuweisung von Dokumenten zu Clustern. Die Berechnung der Bewertungsfunktion für jedes Dokument basiert auf wenigen Feldzugriffen einer vorberechneten Datenstruktur. Die Datenstruktur wird mit einem ausführlichem Korrektheitsbeweis hergeleitet.

Diese Bewertungsfunktion dient als Grundlage für zwei Algorithmen zum Clustern von Dokumenten, die in dieser Arbeit vorgestellt und analysiert werden. Die Algorithmen sind an den bekannten K-Means-Algorithmus angelehnt. Im Kern basieren die Algorithmen auf der gierigen Zuweisung von Dokumenten zu Clustern. Die Qualität der Cluster wird über mehrere Iterationen hinweg verbessert. Nach einer kurzen Analyse der Algorithmen folgen mehrere Optimierungen, die

die Effizienz der Algorithmen noch weiter steigern. Die Optimierungen beinhalten unter anderem eine effizientere Vorberechnung der Datenstrukturen für die Bewertungsfunktion und die Betrachtung der Parallelisierungsmöglichkeiten beider Algorithmen.

Neben den beiden Algorithmen zum Clustern von Dokumenten beleuchtet die Diplomarbeit mehrere Ansätze, bei denen Dokumentencluster verwendet werden, um konjunktive Suchanfragen zu beschleunigen. Der erste Ansatz ist ein neuer Algorithmus, der explizit Cluster benutzt, um schnellere Schnitte zu ermöglichen. Dabei greift der Algorithmus auf einen generischen Schnittalgorithmus für die eigentlichen Schnitte zurück. Dieser Ansatz führt zu einer theoretisch geringeren Zeitkomplexität, wenn die Dokumentencluster gut gewählt sind. Der zweite Ansatz verwendet eine Neuanordnung der Dokumente, um in der Praxis schnellere Laufzeiten zu erreichen. Auch wenn keine theoretisch bessere Zeitkomplexität erreicht wird, so zeigen Experimente, dass dieser Ansatz in der Praxis gut oder sogar besser funktioniert als der erste Ansatz. Die Analyse wird durch die Betrachtung der Einsatzmöglichkeiten dieser Techniken in der Praxis abgerundet.

Im Rahmen der Diplomarbeit entstand ein einfaches, aber funktionales Open-Source-Framework für Volltextsuche in der funktionalen Programmiersprache Haskell. Das Framework enthält auch parallele und effiziente Implementierungen der beiden Clustering-Algorithmen in der Programmiersprache C. Die Experimente dieser Diplomarbeit werden mit Hilfe dieses Frameworks ausgeführt.

Das Clustern von Dokumenten führt bei Experimenten auf realen Datensätzen zu guten Ergebnissen. Evaluiert werden, neben den Laufzeiten der Clustering-Algorithmen, die theoretische Beschleunigung durch die geringere Zeitkomplexität sowie gemessene Beschleunigungen auf realen Suchanfragen. Die Experimente geben einen Einblick in die Beschleunigungen, die in der Praxis erreicht werden können.

Auf dem geläufigem Datensatz GOV2 wird sowohl eine theoretische als auch eine reale Beschleunigung um den Faktor 1,8 erreicht. In der Praxis können also die Schnitte für Suchanfragen fast doppelt so schnell berechnet werden. Auf einem anderen Datensatz, der auf realen Sucheingaben basiert, steigen die gemessenen Beschleunigungen sogar auf den Faktor 4. Experimente zur Laufzeit zeigen, dass auch die Clustering-Algorithmen sehr effizient laufen. Der GOV2-Datensatz kann bei geeigneter Parameterwahl mit einer Geschwindigkeit von 60 µs pro Dokument auf einem handelsüblichen PC geclustert werden. Dabei erreichen sowohl die theoretischen als auch die realen Beschleunigungen für diese Parameter Werte von über 1,75.

Neben den erzielbaren Beschleunigungen beleuchtet die Evaluation der Algorithmen weitere interessante Aspekte und bietet Einblicke in die Einflüsse der verschiedenen Parameter. So wird unter anderem der Einfluss der Größe und Beschaffenheit der Dokumentensammlung auf die Qualität der Clustering-Verfahren untersucht. Untersuchungen zeigen auch, dass für qualitativ hochwertige Ergebnisse nur wenige Wörter in den Algorithmen berücksichtigt werden müssen. Eine kurze Auswertung der produzierten Cluster ergibt, dass die Verfahren es schaffen Terme ungleichmäßig auf die Cluster zu verteilen.

Nach bestem Wissen ist diese Diplomarbeit die erste Arbeit, die ein Verfahren

vorstellt, das explizit die Zeitkomplexität von Schnittalgorithmen durch die Verwendung von Dokumentenclustern verringert. Dies führt zu einer Reihe interessanter und fortführender Fragen, die außerhalb des Rahmens dieser Diplomarbeit liegen. Experimente zeigen, dass das Verfahren im einfachen Fall für konjunktive Suchanfragen gut funktioniert. Weitere Forschung ist nötig um festzustellen, ob und wie das Verfahren auf weitere Bereiche der Volltextsuche übertragen werden kann.

Abstract

Fast and easy access to information has become a keystone in our fast-paced world. Full text search remains an important technique in the area of information retrieval and excels wherever fast access to large amounts of text is a prime concern. Improving full text search is still an active research area. Faster full text search leads to higher throughput, reduced hardware costs and an overall improved user experience through lower latencies.

This thesis focuses on how document clustering can improve the efficiency of conjunctive queries. Conjunctive queries are typically solved by intersecting sets of documents containing the terms of a query. The idea is simple, yet novel: The best set intersection algorithms have a time complexity that is dominated by the size of the smallest set. If the documents can be clustered in a way that each term is frequent in only a few clusters and rare in all the other clusters, an algorithm can process those clusters in which terms occur rarely much faster, and thus improve full text search speed overall.

This idea is incorporated in an efficient scoring function for assigning documents to clusters. The scoring function specifically minimizes the time complexity of conjunctive queries. The scoring function is used by two efficient and scalable clustering algorithms presented in this thesis. Experiments show that clustering documents of the GOV2 data collection can be as fast as 60 µs per document on a single commodity PC.

The clustering algorithms are supplemented by an analysis on how document clusters can be used to improve the performance of conjunctive queries. Two separate techniques are presented. While one leads to better theoretical time complexities, the other can perform better in practice.

The theoretical foundations are backed up by an open source implementation of the clustering algorithms and a small full text search engine framework in the functional language Haskell. The framework is used to conduct experiments on real world datasets.

The experiments show that the approach presented in this thesis works well. On the commonly used GOV2 dataset two-term conjunctive queries run close to 2 times faster and the theoretical time complexity decreases by nearly a factor of 2. For a dataset that consists of documents specialized on a single topic, speedups of over 4 can be measured on actual queries by real users.

Contents

1 Introduction **1**
1.1 Contribution . . . 2
1.2 Outline . . . 2

2 Related Work **3**

3 Technical Foundations **5**
3.1 Full Text Search Basics . . . 5
3.2 Search Engine Design . . . 6
3.3 Conjunctive Queries . . . 7
3.4 Efficient Set Intersections . . . 7
3.5 Compression . . . 9

4 Document Clustering **11**
4.1 Basic Idea . . . 11
4.2 Definitions and Terminology . . . 12
4.3 Fast Scoring Function . . . 12
4.3.1 Scoring Function for Intersections in $\mathcal{O}(m)$ . . . 13
4.3.2 Learning the Query Language Model . . . 20
4.4 Fast Clustering of Documents . . . 21
4.4.1 Fast Multi-Level Clustering . . . 21
4.4.2 Document-Grained vs Iteration-Grained . . . 22
4.4.3 Fast Hierarchical Clustering . . . 23
4.5 Optimizations . . . 24
4.5.1 Parallelization . . . 24
4.5.2 Term Cutoff . . . 25
4.5.3 Omitting α_2 and α_4 . . . 25
4.6 Summary . . . 25

5 Search with Document Clusters **27**
5.1 Set Intersection Data Structure for Clusters . . . 27
5.2 Document ID Relabeling . . . 28
5.3 Hybrid and Parallel Intersections . . . 30
5.4 Distributed Search and Document Clusters . . . 30

6 Implementation **33**
6.1 Core Library . . . 34
6.1.1 Basic Data Types . . . 34
6.1.2 Text Processing . . . 34

6.1.3 Clustering . . . 34
6.1.4 Set Intersections . . . 36
6.2 File Formats . . . 36
6.3 Tools . . . 38
6.3.1 Tokenization . . . 38
6.3.2 Index Construction . . . 38
6.3.3 Clustering . . . 38
6.3.4 Evaluation . . . 38

7 Experimental Evaluation 41
7.1 Evaluation Measures . . . 41
7.2 Experimental Methology . . . 42
7.2.1 Environment . . . 43
7.3 Datasets . . . 43
7.3.1 Document Collections . . . 43
7.3.2 Query Logs . . . 44
7.4 Experimental Results . . . 45
7.4.1 Speedups . . . 45
7.4.2 FMClustering versus TopDown . . . 48
7.4.3 Clustering Effect . . . 50
7.4.4 Compression . . . 51
7.4.5 Speedups with Comparison-Based Intersection Algorithms . . . 52
7.4.6 Query Language Model versus Document Collection Language Model 52
7.4.7 Clustering Stability . . . 53
7.4.8 Influence of Term-Cutoff . . . 54
7.4.9 Influence of Number of Documents . . . 55
7.4.10 Speedups for Single Queries . . . 56
7.4.11 Exact Scoring Function . . . 58
7.5 Clustering Runtimes . . . 58
7.5.1 FMClustering versus TopDown . . . 60
7.5.2 Influence of Term-Cutoff . . . 61
7.5.3 Influence of Document Collection Size . . . 61

8 Conclusion 65
8.1 Summary . . . 65
8.2 Future Work And Outlook . . . 66

Bibliography 67

1. Introduction

Fast and easy access to information has become a keystone in our fast-paced world. With the rise of information technology, the drop in storage costs and the invention of the Internet the amount of available data exploded into inconceivable dimensions. While many methods exist to provide fast and easy access across various domains, full text search remains an important technique in the area of information retrieval. The rise and success of internet search engines make a compelling argument for the adoption of full text search for very large collections of documents. But aside from the Internet, full text search excels wherever fast access to large amounts of text is a prime concern.

While full text search has been heavily studied over the last decades, improving search speed is still a concern and full text search on huge document collections is still a challenge [Con13]. Faster full text searches lead to higher throughput, reduced hardware costs and an overall improved user experience through lower latencies. Furthermore, new and emerging technologies, e. g., question answering or cognitive systems, can benefit from improved full text access.

At the base of every search is the task of finding relevant results. For full text search this corresponds to finding the documents matching a search query (and optionally ranking them). Commonly this task is solved using an *Inverted Index*. For each term, an *Inverted Index* holds a set of documents containing that term. Matching documents to a search query is then simply a matter of intersecting the corresponding sets of documents[1]. However, the process of intersecting sets can involve millions of documents and therefore fast set intersection algorithms are crucial to fast full text search.

This thesis focuses on how document clustering can improve the performance of existing set intersection algorithms. The idea is simple[2]: Efficient set intersection algorithms have an algorithmic time complexity that is dominated by the size of the smallest set. If the documents can be clustered in a way that each term is frequent in only a few clusters and rare in all the other clusters, an algorithm can process those clusters in which terms occur rarely much faster, and thus improve full text search speed overall.

[1] Regarding conjunctive queries, for a more detailed explanation please refer to Section 3.3

[2] This concept is explained in depth in Section 4.1

1.1 Contribution

The contribution of this thesis is manifold:

1. An efficiently computable scoring function for assigning documents to clusters that specifically optimizes the expected time complexity of a query.
2. Two scalable, parallel and efficient clustering algorithms that work well in combination with the scoring function.
3. Two new techniques for using document clusters to accelerate conjunctive queries in full text search.
4. An in-depth evaluation and analysis of theoretical and practical speedups achieved on posting list intersections through document clusterings on real-world datasets using the clustering algorithms discussed in this thesis.
5. A search engine framework implemented in Haskell [JHA+99] that can handle millions of documents consisting of hundreds of gigabytes of text.

1.2 Outline

The rest of the thesis is structured as follows:

- *Chapter 2* looks at related work and discusses previous approaches for both document clustering and improving search performance.
- *Chapter 3* gives a brief overview over the technical foundations of full text search. The discussion is limited to topics relevant to this thesis.
- *Chapter 4* discusses fast clustering of documents. First, an efficient and semantically meaningful scoring function for assigning documents to clusters is introduced and proven. Next, the chapter discusses efficient clustering algorithms. The chapter closes with several optimizations that can improve clustering runtimes.
- *Chapter 5* illustrates how document clusters can be used to accelerate full text search and set intersections. It introduces a new data structure which can be used for conjunctive queries on document clusters and discusses alternative approaches.
- *Chapter 6* explains details of the implementation.
- *Chapter 7* evaluates the clustering algorithms on real world data. A detailed analysis and systematic benchmarks give insights into important parameters.
- *Chapter 8* closes this thesis with a summary and an outlook on further work.

2. Related Work

Many speedup techniques have been studied in the area of full text search, including geographical tiering [CPBY09], static index pruning [DMdSF+05] and dynamic index pruning [Per94]. The techniques improve efficiency by disregarding parts of the index. In contrast, the approach in this thesis achieves speedups while still using the whole index. However, a full evaluation of speedup techniques is out of the scope of this work. The focus here lies on the improvement of search efficiency by clustering documents or partitioning the posting lists.

The idea of clustering documents in the context of full text search is not new. Berkhin [Ber06] gives a good overview over common clustering techniques, many of which can be applied to document clustering.

Many approaches are based on the *clustering hypothesis* [VR79], which states that documents that are relevant to a query tend to be more similar to each other than documents that are non-relevant. Documents similar to each other are clustered together[1]. Queries are then executed via *collection selection*. Document clusters considered irrelevant are disregarded and only clusters relevant to the query are used for searching. This imposes two main problems:

- Which clusters are relevant?
- How many clusters are relevant?

The approach in this thesis circumvents the two problems caused by collection selection by improving search efficiency without losing accuracy.

Clustering Documents

Xu et el. [XC99] use a 2-phase K-means algorithm [M+67] to cluster documents. In a first pass, the first k documents are chosen as initial clusters and all subsequent documents are added to the cluster with the highest similarity. A second pass reassigns documents and corrects errors from the first pass. They use a slightly modified Kullback-Leibler divergence as a pseudo distance metric between clusters and documents which is used to assign the

[1] In this thesis we use *cluster* in the sense of semantic cluster, not computer cluster

documents to clusters. Cluster centers are formed by concatenating all documents in a cluster and treating it as if it where a single document. Collection selection is based on a unigram language model for both queries and clusters. The Kullback-Leibler divergence based on the unigram language model is used to find the top-k clusters for a given query.

More recently, Puppin et al. [PSL06] introduce a method of using co-clustering of documents and queries to perform collection selection. Similar queries and similar documents are clustered together. Each query cluster is assigned a contribution value to each document cluster based on ranking each document in the cluster for each query in the query cluster. For collection selection, the query cluster contribution values are used to score and select the top document clusters. The contribution values are weighted by the similarity of the query to the query clusters.

In her dissertation, Kulkarni [Kul13] reexamines and evaluates how clustering can be used to partition the document collection into shards to maximize both, search efficiency and search effectiveness. The hypothesis is that by clustering documents by similarity, many shards can be disregarded for the query. She introduces a modified version of the K-Means algorithm which improves scalability. The clustering approach is similar to the one in this thesis. Instead of clustering all documents at once, a small sample of documents is picked at random and used to find the initial cluster centers. The rest of the collection is then assigned in a single pass as second step. With a small random sample, K-Means will take significantly less time to converge towards stable clusters. The properties of the clusters are not likely to alter dramatically after assigning the rest of the documents.

All these approaches concentrate on collection selection by clustering similar documents together. However, they suffer from the aforementioned limitations of determining which clusters are relevant. This thesis breaks out in a new direction and tries to improve efficiency not by eliding clusters, but by organizing clusters in a way so that algorithms will run faster. To the best of our knowledge, this thesis is the first to look at this question. However, ideas of clustering algorithms, using query logs, and other techniques still apply and are used in this thesis.

Improving Search Efficiency

Apart from collection selection, there exist other methods to improve search efficiency. For term partitioned indexes, Luccese et al. [LOPS07] formulate term partitioning as an optimization problem. Their algorithm tries to minimize the average latency for queries and maximizes overall throughput. Probabilities of terms occurring in queries are explicitly used in the optimization problem. They present a greedy algorithm which tries to assign terms to servers and use query logs to estimate the query probabilities. This resembles closely the goal of this thesis, but is limited to term partitioned indexes. The work in this thesis is independent from a concrete partitioning scheme and can be applied to term and document partitioned indexes.

3. Technical Foundations

This chapter gives a brief overview over the fundamental concepts and data structures for full text search. The explanations in this chapter are kept short and simple and only go into depth when relevant to this thesis.

3.1 Full Text Search Basics

Full text search has been applied to many domains, most notably web search [BP98]. While there exist other data structures, inverted indexes have become the prevalent choice for full text search due to their good runtime characteristics, compact representation and simple structure. An extensive overview over full text search is given in the introductory book *Modern Information Retrievel* by Baeza-Yates and Ribeiro-Neto [BYRN+99].

An inverted index maps terms to their occurrences in documents as illustrated in Figure 3.1. An inverted index holds, for each *term*, a list of all occurrences in the documents, commonly referred to as *posting list*. For compact representation, unique identifiers are used to represent terms (abbreviated *term ID*) and documents (abbreviated *doc ID*). A dictionary maps words to their corresponding term IDs. The posting list can, depending on the requirements, hold additional information, e. g., position and relevance of the term within the document.

Before a document is indexed it is preprocessed. Common steps include stripping any unwanted information, e. g., potential markup, and removing term separators (spaces, punctuation marks, ...). Removing common words with low semantic value, called *stop words*, can reduce the index size and improve search effectiveness. *Stemming*, the process of transforming words into their word stem, can be beneficial at times.

The preprocessing step returns a list of terms for a document. This list can then be used to easily add the document to the inverted index. Typically the whole inverted index containing all documents is constructed in an offline preprocessing phase.

Listing all the documents containing a particular term is a simple lookup in the inverted index. An implementation for accessing posting lists should be optimized for speed to allow for fast searches.

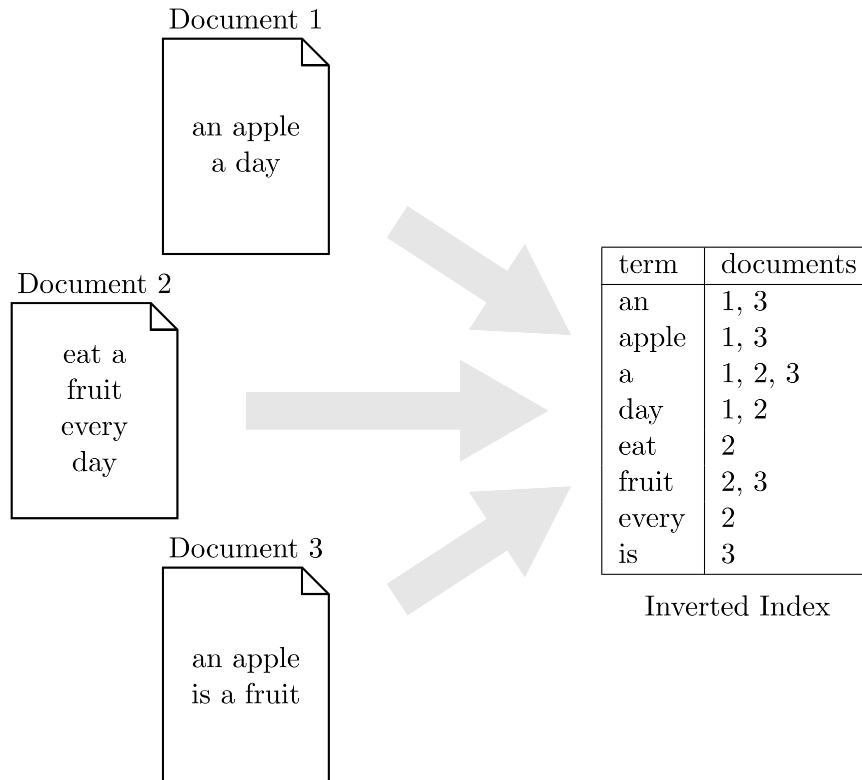

Figure 3.1: Inverted Index

3.2 Search Engine Design

Large-scale enterprise search engines require a carefully thought out design. For huge document collections like the web, single-server, *monolithic* search engines do not suffice [BDH03]. For one thing, the index does not fit into either internal or external memory. By partitioning the index and distributing it over several *shards* memory consumption for a single server can be reduced and throughput and latency can then easily be improved through parallelization and concurrency. Replication of shards can further improve search engine performance through inter-query parallelization, answering multiple queries concurrently.

A typical simple setup for a shard-based search engine is illustrated in Figure 3.2. Queries are handled by a broker which in return forwards the queries to relevant shards. The broker then joins the results from each shard into a single, final result.

There are two common partitioning schemes: by term and by document [JO95]. When using partitioning by term, an inverted index is built for the entire collection and each shard holds only a subset of terms of an inverted index. Queries are only sent to the shards containing a term in the query. For conjunctive queries, if terms are distributed among shards, posting lists have to be sent between shards causing significant communication overhead. When partitioned by document, each shard holds a subset of documents and holds a full inverted index for those documents. In this case, the queries are sent to all shards by the broker.

Although term partitioning seems to be the more efficient choice, since only some of the shards have to be contacted for every query, document partitioning has shown to have

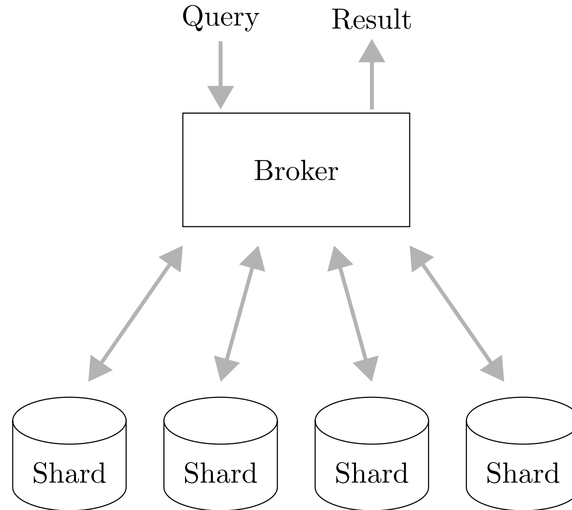

Figure 3.2: Search Engine Architecture

better characteristics in practice. Term partitioning suffers from communication overhead and load imbalancing problems [MMR00].

Traditionally, inverted indexes have been stored on hard disks due to the size and cost of random access memory. More recently, with the decrease of internal memory costs, inverted indexes stored in internal memory have become more popular [Tra10]. Because disk access and seek times are order of magnitudes slower than RAM access, storing inverted indexes in internal memory can have a tremendous impact on the performance of a search engine.

3.3 Conjunctive Queries

While inverted indexes can also be used to answer *phrase queries* and *disjunctive queries*, this thesis concentrates on *conjunctive queries*. *Conjunctive queries*, also referred to as *boolean AND queries*, select documents that contain **all** the terms specified in the query, hence the word *conjunctive*. They are generally faster to compute than *disjunctive* or *phrase queries* and therefore are predominant in the area of web search where the document collections are generally large and a fast query processing time is important.

Evaluating *conjunctive queries* boils down to simply intersecting the posting lists for all terms specified in the query. Efficient algorithms are explained in detail in the next section.

For an inverted index residing in external memory the posting lists for the terms in a query are loaded into memory and intersection is then performed in main memory. Due to slow access and high latency for external memory, loading times are generally much higher than intersection run times and optimizing disk accesses is a bigger concern than improving intersection algorithms. However, posting lists might be cached and in in-memory search engines, intersection run times play a crucial role for latency and throughput. This thesis assumes posting lists in memory.

3.4 Efficient Set Intersections

A posting list can effectively be seen as a set of integers. Intersections of sets of integers have been thoroughly researched in the last decades and numerous efficient algorithms

have been proposed. Algorithms relevant to this thesis are presented and discussed here. All algorithms require a precalculated data structure for efficient intersection. For the remainder of this section n denotes the size of the larger set and m the size of the smaller set.

While there exist specialized solutions for intersecting multiple sets, only two set intersections are considered in this section. Extension to multiple terms can easily be achieved by first intersecting the two smallest sets, and iteratively intersecting the result with the next larger set.

Zipper

Consider sorted posting lists. A trivial, yet effective solution is to simultaneously scan both lists in order, advancing in the list with the currently smaller element. If the current elements in both lists are equal, it can be added to the result set.

Zipper has an algorithmic complexity of $\mathcal{O}(n + m)$ with a small constant factor. It is a very good solution when both sets are of about equal size.

Lookup

Sanders and Transier [ST07] introduce an efficient set intersection algorithm based on buckets. Elements are divided into buckets. A top-level data structure holds the sizes and pointers to a bottom-level data structure containing the actual elements for each bucket. These combined data structures are referred to as *Lookup list*.

The upper bits of an element are used to determine the index of the bucket it is placed in. This can be achieved efficiently with machine bit-shifting instructions. A parameter B controls the average bucket size and together with the set size determines how many upper bits are used. The number of bits also dictate how many buckets exist. Elements in each bucket are sorted.

To intersect two sets, the smaller set is scanned in order. For each element in the smaller set, the bucket index in the larger set is computed. A lookup in top-level data structure yields the starting position of that bucket in the bottom-level data structure. The elements of the bucket are scanned in order until one of the following occurs:

- The end of the bucket is reached, in which case the element from the smaller set is not contained in the larger set.
- A larger element in the bucket is reached. Because the elements in the buckets are ordered the element is not contained in the larger set.
- The element is found in the bucket.

Because elements in both sets are sorted, sequential scans starting in the same bucket can advance from the previous position in the bottom-level data structure. This ensures that each element of a bucket is scanned at most once.

Sanders and Transier show that, assuming randomly distributed elements, bucket sizes average about B and the algorithmic complexity is $\mathcal{O}(m + \min(n, Bm))$. With small bucket sizes this can effectively be seen as $\mathcal{O}(m)$. Lookup is suitable where large ratios between set sizes are to be expected.

3.5 Compression

Inverted indexes can use significant amounts of memory. Compression helps to keep the memory requirements small. Moreover, when inverted indexes reside in external memory, compression improves the number of posting lists that can be cached as well as the transfer time from external to internal memory for the posting lists.

A simple approach is to use Δ-*encoding* together with *variable-length encoding.* Instead of storing the integers, respectively the document IDs directly, Δ-encoding stores the differences between the integers. That is, the sorted sequence $d_1, \ldots, d_n$ is stored as $d_1, d_2 - d_1, d_3 - d_2, \ldots, d_n - d_{n-1}$. Because variable-length encoding uses less bits for smaller integers, less bits are needed to encode a posting list in total if Δ-encoding is used.

Common variable-length encodings include *Golomb coding* [Gol66], *Elias-*γ and *Elias-*δ encoding [Eli75]. A detailed explanation is out of the scope of this thesis. However, Elias-γ coding requires $1 + 2\lfloor \log_2 x \rfloor$ bits to encode an arbitrary integer $x > 0$, whereas Elias-δ coding uses $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$ bits [BYRN+99]. Golomb coding requires an additional parameter b. With a good choice for b and values with a geometric distribution, Golomb coding uses roughly about $2 + \log_2 x$ bits on average.

4. Document Clustering

4.1 Basic Idea

The idea behind this thesis is simple, yet novel. Assume documents can be clustered in a way that each term occurs only in a minority of clusters. In other words, documents containing a particular term only occur in a few clusters. Then, for conjunctive queries, clusters can be elided when they do not contain any documents for any term in a query.

While such a clear cut clustering is not always possible for realistic inputs, clustering documents can still have great impact on intersection performance. The goal is to cluster documents so that the intersections within the clusters run faster in total than intersecting the whole posting list.

The advantages are illustrated using a simple example. Assume a conjunctive query with two terms and four document clusters. Table 4.1 lists the number of documents containing term 1 and 2 in each cluster and in total. As described in Section 3.4, modern and efficient set intersection algorithms have a time complexity that is dominated by the smaller set. In this case we assume the linear complexity of *Lookup*. For the normal approach without clusters, the smaller set has a size of 37 000 and thus intersecting both sets will result in a time complexity of approximately 37 000. If each cluster is intersected separately, the intersection time complexities will be approximately 2 000, 1 000, 1 000, 1 000 for clusters 1 through 4 respectively.

Cluster	#Documents Term 1	#Documents Term 2
1	2000	10 000
2	10 000	1000
3	40 000	1000
4	1000	25 000
Total	53 000	37 000

Table 4.1: Example clustering of documents

Added up this totals a time complexity of approximately 5 000, nearly an order of magnitude faster compared to the original 37 000!

Finding a clustering that works well for all or most combinations of terms, not only a single pair of terms, is a challenge. However this thesis shows that the idea works in principle and on real data and illustrates its further benefits, e. g., compression. This chapter discusses algorithms, data structures and optimizations to cluster documents in an effective and efficient way.

4.2 Definitions and Terminology

This section introduces some terminology and definitions used in the remainder of this thesis.

Let $D = \{d_1, \ldots, d_n\}$ be the set of all documents and $T = \{t_1, \ldots, t_m\}$ the set of distinct terms occurring in documents from D. A document $d_i = t_{i1}t_{i2} \ldots t_{ij}$ is a list of terms. For simplicity $t \in d_i$ is used to refer to terms occurring in d_i. The indices are omitted whenever the context is clear. The size of a document $d \in D$ is denoted by $|d|$. Note that terms might occur more than once in a document.

A clustering is given by the set $C = \{c_1, \ldots, c_k\}$. A cluster $c_i \in C$ is a set of documents $c_i = \{d_{i1}, \ldots, d_{il}\}$. For this thesis, clusters are non-overlapping: $\forall i, j, i \neq j : c_i \cap c_j = \emptyset$. Indices are again omitted whenever the context is clear.

For this thesis the universe of queries $Q = \{\{t, u\} \mid t, u \in T, t \neq u\}$ is limited to the set of all possible two term queries. Note that there is no order for the terms in queries and queries containing the same term twice are prohibited. This is not a serious constraint, as conjunctive queries are commutative and intersecting a set with itself is the identity function.

The number of documents containing a term in a cluster is given by $n(c, t) = |\{d_i \mid t \in d_i, d_i \in c\}|$. This can also be viewed as the size of the posting list for a term t when considering only documents $d \in c$ from a single cluster c. Similarly the number of clusters containing at least one document with the term t is denoted by $k(t) = |\{c_i \mid \exists d \in c_i : t \in d\}|$

Iverson brackets [Knu92] are used to increase readability. Given a logical statement P the brackets are used as follows.

$$[P] = \begin{cases} 1, & \text{if } P \text{ is true.} \\ 0, & \text{otherwise.} \end{cases}$$

At times it is convenient if T is totally ordered. Given a total order and two terms $t, u \in T$, $t < u$ is true if, and only if, t precedes u.

4.3 Fast Scoring Function

While many other approaches cluster documents by similarity, this thesis runs another path. The documents are clustered by explicitly optimizing the expected query time. To keep things simple, the query time is modeled by the set intersection complexity. This might be oversimplifying, however, optimizing the set intersection complexity will likely also reduce query time in general, despite constant factors and overhead. Only two term conjunctive queries are considered here. The usual approach for queries with more terms is to start by intersecting the two smallest sets of the query. Therefore the effects for two term queries are likely to also apply to queries with more terms.

The following paragraphs introduce a cost function that models the time complexity for two term queries. The cost function uses the assumption that all clusters are intersected independently. The complexity of determining the clusters containing documents for any term for a query is added to the total complexity of intersecting all the posting lists in all those clusters.

Let $\Phi(n,m)$ be the time complexity of intersecting two sets with size n and m, C a clustering of documents and C_q be the set of clusters with at least one document containg any term from the query $q \in Q$. We use the function Φ to model a cost function $\Psi_C(t,u)$ for a query $\{t,u\} \in Q$:

$$\Psi_C(t,u) = \Phi(k(t),k(u)) + \sum_{c_q \in C_q} \Phi\left(n(c_q,t), n(c_q,u)\right) \tag{4.1}$$

Note that C_q is defined implicitly by t and u.

The cost function Ψ_C is justified by the following observations: The number of clusters containing term t and u is $k(t)$ and $k(u)$ respectively. The set of clusters C_q containing documents for a query $q \in Q$ can be determined using standard set intersections in $\Phi(k(t),k(u))$. For this, each term has a *cluster posting list* holding the cluster IDs containing at least one document with that particular term. Iterating over each cluster, the set of documents for each cluster can be computed in $\Phi(n(c_q,t),n(c_q,u))$ and added to the output set.

Let Q be the sample space of all queries $q \in Q$ and $p(t \wedge u) \coloneqq p(\{t,u\})$ the probability mass function for a query $q = \{t,u\} \in Q$. Given a clustering C, Ψ_C is a random variable and has the expected value of:

$$\begin{aligned}
\mathbb{E}\left[\Psi_C\right] &= \sum_{\{t,u\} \in Q} p\,(t \wedge u)\, \Psi_C(t,u) \\
&= \sum_{\{t,u\} \in Q} p\,(t \wedge u) \left(\Phi\left(k(t),k(u)\right) + \sum_{c_q \in C_q} \Phi\left(n(c_q,t), n(c_q,u)\right) \right)
\end{aligned} \tag{4.2}$$

The expected value for Φ_C is a meaningful cost function for an optimization problem. The work, that is the time complexity, should be minimal over all queries, with more attention to more frequent queries. This also permits defining a simple scoring function for local search clustering algorithms. Let C' be the clustering after adding document d to cluster $c \in C$. The scoring function then is:

$$\mathrm{Score}(d,c,C) \coloneqq \mathbb{E}[\Psi_{C'}] - \mathbb{E}[\Psi_C] \tag{4.3}$$

As C is mostly defined by the context, we use $\mathrm{Score}(d,c)$ as short for $\mathrm{Score}(d,c,C)$. The next sections discuss how $\mathrm{Score}(d,c)$ can be computed efficiently. Note that by definition a lower score means a better score.

4.3.1 Scoring Function for Intersections in $\mathcal{O}(m)$

The best set intersection algorithms have a time complexity of $\mathcal{O}(m)$, as discussed in Section 3.4, where m is the size of the smaller set. This allows a simple approximation for

$\Phi(n, m) \coloneqq \min(n, m)$. While there exist different constant factors for set sizes and ratios, this allows for an easier analysis and still holds true in principle: The smaller the smaller set, the faster the intersection will be.

This section shows how to compute the scoring function $\text{Score}(d, c)$ in $\mathcal{O}(|d|)$ time. The function is split into four separate parts $\alpha_i(d, c), i \in \{1, 2, 3, 4\}$, of which each can be computed separately in linear time $\mathcal{O}(|d|)$.

Lemma 4.3.1. *The function* $\text{Score}(d, c)$ *can be rewritten using an arbitrary total order on* T *as:*

$$\text{Score}(d, c) = \alpha_1(d, c) + \alpha_2(d, c) + \alpha_3(d, c) + \alpha_4(d, c)$$

where

$$\begin{aligned}
\alpha_1(d, c) &= \sum_{t \in d} \sum_{u \in T} p(t \wedge u)[n(c, t) < n(c, u)] \\
\alpha_2(d, c) &= \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)[n(c, t) = n(c, u)] \\
\alpha_3(d, c) &= \sum_{t \in d} \sum_{u \in T} p(t \wedge u)[k(t) < k(u) \wedge t \notin c] \\
\alpha_4(d, c) &= \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)[k(t) = k(u) \wedge t, u \notin c]
\end{aligned}$$

Proof. Let $c' = c \cup \{d\}$ be the new cluster after adding document d. For simplicity we write $n'(c, t) \coloneqq n(c', t)$ and $k'(t)$ when referring to the new clustering.

$$\begin{aligned}
\text{Score}(d, c) &= \mathbb{E}[\Psi_{C'}] - \mathbb{E}[\Psi_C] \\
&= \sum_{\{t,u\} \in Q} p\,(t \wedge u) \left(\min\left(k'(t), k'(u)\right) + \sum_{c_q \in C'_q} \min(n'(c_q, t), n'(c_q, u)) \right) \\
&\quad - \sum_{\{t,u\} \in Q} p\,(t \wedge u) \left(\min\left(k(t), k(u)\right) + \sum_{c_q \in C_q} \min(n(c_q, t), n(c_q, u)) \right) \\
&= \sum_{\{t,u\} \in Q} p(t \wedge u) \sum_{c_q \in C'_q} \min(n'(c_q, t), n'(c_q, u)) \\
&\quad - \sum_{\{t,u\} \in Q} p(t \wedge u) \sum_{c_q \in C_q} \min(n(c_q, t), n(c_q, u)) \\
&\quad + \sum_{\{t,u\} \in Q} p(t \wedge u) \min(k'(t), k'(u)) \\
&\quad - \sum_{\{t,u\} \in Q} p(t \wedge u) \min(k(t), k(u))
\end{aligned}$$

As all but one cluster are left unchanged by the document addition, it is obvious that $\forall c_q \neq c : n'(c_q, t) \equiv n(c_q, t)$ and therefore the sum over the clusters can be simplified to

$$
\begin{aligned}
&= \sum_{\{t,u\}\in Q} p(t \wedge u) \left(\underbrace{\min(n'(c,t), n'(c,u)) - \min(n(c,t), n(c,u))}_{a(t,u)} \right) \\
&\quad + \sum_{\{t,u\}\in Q} p(t \wedge u) \left(\underbrace{\min(k'(t), k'(u)) - \min(k(t), k(u))}_{b(t,u)} \right) \\
&= \underbrace{\sum_{t\in T} \sum_{\substack{u\in T \\ u>t}} p(t \wedge u) a(t,u)}_{(1)} + \underbrace{\sum_{t\in T} \sum_{\substack{u\in T \\ u>t}} p(t \wedge u) b(t,u)}_{(2)} \qquad (4.4)
\end{aligned}
$$

Note that by definition of Q and with the total ordering on T the sum $\sum_{\{t,u\}\in Q}$ can be rewritten as $\sum_t \sum_{u>t}$. The notation $t \in T$ is omitted for simplicity. In the following the proof only handles (1). The proof for (2) is analogous with the additional constraint of $t \notin c$ for most cases.

Obviously $n'(c,t) = n(c,t) + \delta_t$ where

$$
\delta_t = \begin{cases} 1, & \text{if } t \in d \\ 0, & \text{otherwise} \end{cases}
$$

and so $a(t,u)$ can be defined as follows:

$$
\begin{aligned}
\zeta(t,u) &\coloneqq \delta_t [n(c,t) < n(c,u)] \\
\eta(t,u) &\coloneqq \delta_u [n(c,u) < n(c,t)] \\
\xi(t,u) &\coloneqq \min(\delta_t, \delta_u) [n(c,u) = n(c,t)] \\
a(t,u) &= \min(n(c,t) + \delta_t, n(c,u) + \delta_u) - \min(n(c,t), n(c,u)) \\
&= \begin{cases} \delta_t, & \text{if } n(c,t) < n(c,u) \\ \delta_u, & \text{if } n(c,u) < n(c,t) \\ \min(\delta_t, \delta_u), & \text{if } n(c,t) = n(c,u) \end{cases} \\
&= \zeta(t,u) + \eta(t,u) + \xi(t,u)
\end{aligned}
$$

Without loss of generality assume a total order where $\forall t, u \in T : t \leq u \implies n(c,t) \leq n(c,u)$. Then (1) from Equation 4.4 can be rewritten as

$$
\begin{aligned}
(1) &= \sum_t \sum_{u>t} p(t \wedge u) a(t,u) \\
&= \sum_t \sum_{u>t} p(t \wedge u) \left(\zeta(t,u) + \underbrace{\eta(t,u)}_{\equiv 0} + \xi(t,u) \right) \\
&= \sum_t \sum_{u>t} p(t \wedge u) \zeta(t,u) + \sum_t \sum_{u>t} p(t \wedge u) \xi(t,u)
\end{aligned}
$$

Because $t \notin d \implies \delta_t = 0 \implies \forall u : \zeta(t,u) = 0$ the outer sum in the first summand can be limited to terms from the document and similarly because $t \notin d \vee u \notin d \implies \xi(t,u) = 0$ the second summand can be limited to terms from the document in inner and outer sums

$$= \sum_{t \in d} \sum_{u > t} p(t \wedge u)\zeta(t,u) + \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)\xi(t,u)$$

From the definition of ζ and the ordering, it follows that $u < t \implies \zeta(t,u) = 0$ and therefore the inner sum can be expanded from the range $u \in T : u > t$ to the whole range of terms $u \in T$

$$= \sum_{t \in d} \sum_{u \in T} p(t \wedge u)\zeta(t,u) + \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)\xi(t,u)$$

Evidently because $t \in d \implies \forall u \in T : \zeta(t,u) = [n(c,t) < n(c,u)]$ and $t, u \in d \implies \xi(t,u) = [n(c,t) = n(c,u)]$ this can be rewritten using the definitions from the lemma

$$\begin{aligned} &= \sum_{t \in d} \sum_{u \in T} p(t \wedge u)[n(c,t) < n(c,u)] \\ &\quad + \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)[n(c,t) = n(c,u)] \\ &= \alpha_1(d,c) + \alpha_2(d,c) \end{aligned}$$

While the definition from the lemma uses an arbitrary total order, the proof adds an additional constraint to the ordering. This constraint however is not a problem in practice, because the constraint allows an arbitrary order between terms $t, u \in T : n(c,t) = n(c,u)$.

□

Using a Unigram Language Model for Queries

A language model tries to estimate the probabilities of sentences. Sentences are modeled as a list of terms, e. g., $t_1 t_2 t_3$. In the context of conjunctive two-term queries, the language model should estimate the probability of a query. Since conjunctive queries are commutative, the language model should ignore the order of the terms, that is $p(t_1 \wedge t_2) \coloneqq p(t_1 t_2) = p(t_2 t_1)$. One of the simplest language models is a unigram language model. In a unigram language model the probability of a term occurring is independent from other terms.

So far the analysis assumes that $p(t \wedge u)$ is known. However in practice this will not be the case. For one thing, determining $p(t \wedge u)$ empirically is mostly infeasible. The set of all two term queries can be prohibitively large as its size is quadratic in the number of terms. In a unigram language model for two-term queries the probability of a term occurring in a query is independent from the other term. In other words, in a unigram language model the probability for two terms t and u occurring is $p(t \wedge u) = p(t)p(u)$. The unigram language model allows to calculate the scoring function very efficiently, as is shown later on. It is important to see that the unigram language model is only used for the scoring function itself.

While a unigram language model for queries might be oversimplifying, there is still a justification for its use. A simple example illustrates how correlation between terms can

still be preserved even if it is not directly reflected in the scoring function. This is because the scoring function takes all terms of a document into account.

Consider the following simple example. Assume the terms t_1 and t_2 occur frequently in general, but rarely together in either queries or documents. Let cluster c_1 hold the documents where only term t_1 occurs and cluster c_2 the documents where only term t_2 occurs. Let documents with both terms t_1 and t_2 be evenly distributed over both clusters. Assume a new document in which term t_1 occurs, but term t_2 does not. Obviously c_1 is the better choice for this document. This will also be reflected in the scoring function. Because $n(c_1, t_1) > n(c_1, t_2)$ it follows that $[n(c_1, t_1) < n(c_1, t_2)] = 0$ for cluster c_1, whereas $n(c_2, t_1) < n(c_2, t_2)$ and therefore $[n(c_2, t_1) < n(c_2, t_2)] = 1$ for cluster c_2. With all else equal, this leads to a lower and therefore better score for cluster c_1, just as desired. Even with a unigram language model for the scoring function, correlation between terms will be reflected in the clusters and therefore in the scoring function.

Calculating $\alpha_1(d, c)$ and $\alpha_3(d, c)$ in $\mathcal{O}(|d|)$

With the unigram language model, α_1 and α_3 from Lemma 4.3.1 can be calculated using lookup tables as formulated in the following lemma.

Lemma 4.3.2. *Using a unigram language model, α_1 and α_3 can be calculated using lookup tables L_1^c and L_3.*

$$\alpha_1(d, c) = \sum_{t \in d} p(t) L_1^c[t]$$
$$\alpha_3(d, c) = \sum_{\substack{t \in d \\ t \notin c}} p(t) L_3[t]$$

Proof. The proof is outlined here for α_1, but also applies to α_3.

$$\begin{aligned}
\alpha_1(d, c) &= \sum_{t \in d} \sum_{u \in T} p(t \wedge u)[n(c, t) < n(c, u)] \\
&= \sum_{t \in d} \sum_{u \in T} p(t) p(u)[n(c, t) < n(c, u)] \\
&= \sum_{t \in d} p(t) \sum_{u \in T} p(u)[n(c, t) < n(c, u)] \\
&= \sum_{t \in d} p(t) \sum_{\substack{u \in T \\ n(c,t) < n(c,u)}} p(u)
\end{aligned}$$

Let be L_1^c be a lookup table with

$$L_1^c[t] := \sum_{\substack{u \in T \\ n(c,u) > n(c,t)}} p(u)$$

which proves the lemma

$$\alpha_1(d, c) = \sum_{t \in d} p(t) L_1^c[t]$$

The same applies for α_3 (without proof):

$$\alpha_3(d,c) = \sum_{\substack{t \in d \\ t \notin c}} p(t) L_3[t]$$

where

$$L_3[t] \coloneqq \sum_{\substack{u \in T \\ k(u) > k(t)}} p(u)$$

$\square$

Note that L_1^c has to be calculated for each cluster c individually while L_3 can be calculated for all clusters globally. For each term t, the lookup tables L_1^c and L_3 hold the prefix sum $\sum_{u>t} p(u)$ with the terms ordered by $n(c,t)$ and $k(t)$ respectively. Lemma 4.3.2 directly suggests a simple algorithm to calculate L_1^c for a cluster c:

1. Sort all t by $n(c,t)$ in descending order
2. Calculate the prefix sum $L_1^c[t]$ with a linear scan through the terms.

Equivalently the same applies to L_3, however the elements are sorted using $k(t)$ instead of $n(c,t)$. With the lookup table it is evident that α_1 and α_3 can be calculated in $\mathcal{O}(|d|)$.

Efficiently sorting the terms by their frequencies can turn into a complex matter. When clusters do not contain many documents and therefore the frequencies are generally low, a radix sort or counting sort can sort the terms in $\mathcal{O}(|T|)$. However, later on the frequencies can become significantly larger than the number of terms $|T|$. If terms are already nearly sorted from the previous calculation of L_1^c or L_3 an insertion sort can be an efficient alternative in $\mathcal{O}(|T|)$. In any case the sorting of terms can be achieved in $O(|T| \log |T|)$ using efficient comparison based sorting algorithms. The further analysis assumes that the frequencies are bound by an upper constant limit and sorting can be achieved in $\mathcal{O}(|T|)$.

Calculating $\alpha_2(d,c)$ and $\alpha_4(d,c)$ in $\mathcal{O}(|d|)$

Lemma 4.3.3. *Assuming statistical independence α_2 and α_3 can be calculated using lookup tables*

$$\alpha_2(d,c) = \sum_{t \in d} p(t) L_2[t]$$
$$\alpha_4(d,c) = \sum_{\substack{t \in d \\ t \notin c}} p(t) L_4[t]$$

Proof. As with the previous proofs, the proof only handles the case of α_2. The proof for α_4 is analogous.

$$\begin{aligned}\alpha_2(d,c) &= \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t \wedge u)[n(c,t) = n(c,u)] \\ &= \sum_{t \in d} \sum_{\substack{u \in d \\ u > t}} p(t)p(u)[n(c,t) = n(c,u)] \\ &= \sum_{t \in d} p(t) \sum_{\substack{u \in d \\ u > t}} p(u)[n(c,t) = n(c,u)] \\ &= \sum_{t \in d} p(t) \sum_{\substack{u \in d \\ u > t \\ n(c,t) = n(c,u)}} p(u)\end{aligned}$$

Once again let L_2 be a lookup table with

$$L_2[t] \coloneqq \sum_{\substack{u \in d \\ u > t \\ n(c,t) = n(c,u)}} p(u)$$

which proves the lemma

$$\alpha_2(d,c) = \sum_{t \in d} p(t) L_2[t]$$

□

It remains to show how L_2 and L_4 can be calculated efficiently. The algorithm for calculating L_2 is illustrated in Algorithm 1.

Algorithm 1 L_2(d,c)

```
1: H ← uniquesort(d)
2: l ← |H|
3: currentCount ← −1
4: for i ∈ 0, . . . , l − 1 do
5:     t ← H[i]
6:     if n(c, t) ≠ currentCount then
7:         sum ← 0
8:         currentCount ← n(c, t)
9:     end if
10:    L2[i] ← sum
11:    sum ← sum + p(t)
12: end for
13: return L2
```

H is the array of terms from document d sorted by frequency $n(c,t)$ with duplicate terms removed. Since the order of terms is not fixed beforehand, without loss of generality assume an ordering where $u > t \implies t$ appears after u in H. This allows a linear scan through H to build L_2. L_2 is a prefix sum of $p(t)$ using $n(c,t)$ as sort key.

The definition here for L_2 differs from the definition in the lemma, since L_2 is indexed over the position of terms in H, not the terms itself. However H can be used to calculate α_2:

$$\alpha_2(d,c) = \sum_{i=0}^{i<|H|} p(H[i])L_2[i]$$

Note that L_2 and L_4 are unique to each cluster c and document d. Therefore efficient calculation in $\mathcal{O}(|d|)$ for both lookup tables is necessary. It is clear that lines 2–13 can be calculated in $\mathcal{O}(|H|) \subset \mathcal{O}(|d|)$. However sorting in $O(|d|)$ remains difficult as the range of $n(c,t)$ is potentially much larger than $|d|$, rendering linear sorting algorithms infeasible. However no absolute order between terms is necessary. The algorithm only needs groups of terms with the same value for $n(c,t)$. Using a hash table based method, these groups can easily be calculated in $\mathcal{O}(|d|)$ and allows uniquesort to be implemented in $\mathcal{O}(|d|)$.

L_4 can easily be produced with the same algorithm by additionally filtering out all terms $t \notin c$ in H. This can be achieved with a linear scan through d in $\mathcal{O}(|d|)$.

Summary

Theorem 4.3.4. *Using the pre-calculated arrays L_1^c and L_3, the probability function $p(t)$ and using a unigram language model for queries, the scoring function $Score(d,c)$ can be calculated in $O(|d|)$.*

Proof. This follows directly from Lemma 4.3.1, Lemma 4.3.2 and Lemma 4.3.3. □

From Theorem 4.3.4, it follows that the scoring function defined in Equation 4.3 can be calculated efficiently. It provides a semantically meaningful and fast scoring function which can be used in local search clustering algorithms. The issue of pre-calculating L_1^c and L_3 is discussed in detail in Section 4.4.2.

4.3.2 Learning the Query Language Model

The scoring function requires the probability $p(t)$ of a term occurring in a query to be known in advance. Therefore a method for determining the probabilities is necessary. A common solution when using a unigram language model is to use the maximum likelihood estimator [SC99].

Query Log

If query logs are present the solution is straightforward. Let $n_\Upsilon(t)$ be the number of occurrences of a term t in a query log Υ. The probability of a term occurring in a query can then be determined using a maximum likelihood estimator

$$p_{ML}(t) = \frac{n_\Upsilon(t)}{\sum_{u \in T} n_\Upsilon(u)}$$

Note that there is no necessity to use additive smoothing or similar measures [CG96] here, as terms with zero probability are not a problem. Smoothing reduces noise and in the case of additive smoothing prevents terms with zero probability. Documents consist of many, possibly correlated, terms. Even if some terms have an estimated probability of zero, other

terms will contribute to the scoring function. Furthermore terms with no occurrences in the query log will have a small impact on the scoring function even with additive smoothing. This is because $p(t)$ is low for terms with no occurrences even with additive smoothing and terms with low $p(t)$ have little impact on the scoring function. Finally, scoring a single document wrongly does not have a big impact on the total running time of a query.

Document Collection

If a query log is not present, some other method is required to estimate $p(t)$. A simple presumption is that queries and the document collection share the same language model. This leads to similar definition as before. Let $tf(t)$ be the number of occurrences of a term t in the document collection D. The maximum likelihood estimator is given by

$$p_{ML}(t) = \frac{tf(t)}{\sum_{u \in T} tf(u)}$$

The assumption that the document collection and queries will share the same language model is somewhat bold. However, in absence of any other information about queries, the document collection language model can lead to reasonable results as shown in Section 7.4.6.

4.4 Fast Clustering of Documents

While the previous section focuses on how to score documents efficiently, this section discusses efficient clustering algorithms. To be applicable in the context of full-text search, clustering algorithms have to be fast and scalable. Algorithms with non-linear time or space complexities become less feasible as the amount of documents increases.

4.4.1 Fast Multi-Level Clustering

Presented here in Algorithm 2 is an iterative method loosely inspired by the K-Means Algorithm [M$^+$67] and modified to improve performance over the standard algorithm. Though it is similar to the algorithm presented by Kulkarni [Kul13] it was developed independently. While the algorithm from Kulkarni takes one sample and then runs one assignment round, the algorithm presented here recursively samples the documents and runs an assignment round in each iteration.

The hypothesis behind the algorithm is that clustering a small random subsample of documents will give a good initial solution which can be further refined after adding the rest of the documents. Finding good solutions for a considerably smaller subset is likely to be much more efficient.

The algorithm recursively extracts a random subset of documents until the amount of remaining documents is equal to the amount of clusters. Each document is then treated as a cluster for the initial solution. The documents in each recursion step are then iteratively reassigned to the best fitting cluster until no significant improvement occurs. This corresponds to the iterations in the K-Means algorithm. By recursively using less documents, the algorithm can spend more time on deeper recursion levels, where solutions are likely to fluctuate and finding good clusters is important. On higher recursion levels the algorithm can spend less time where a stable solution is more likely and simply assigning documents suffices.

Algorithm 2 FMClustering

Require: documents D, number of clusters K, shrink factor SF

1: **if** $|D| = K$ **then**
2: **return** $\{\{d_1\}, \{d_2\}, \ldots, \{d_K\}\}$
3: **end if**
4: $n \leftarrow \max(k, SF \cdot |D|)$
5: $D' \leftarrow$ random subsample of size n taken from D
6: $C \leftarrow$FMClustering(D', K, SF)
7: **repeat**
8: **for all** $d \in D$ **do**
9: assign d to the cluster $c_d \in C$ where d fits "best"
10: **end for**
11: **until** no "significant" improvement

The algorithm leaves two issues open. Firstly, documents are added to the cluster that fits "best". The best cluster $\hat{c}$ for a document d is determined using the scoring function from Section 4.3:

$$\hat{c} = \arg\min_{c \in C} \text{Score}(d, c)$$

Using this definition, documents are explicitly clustered to minimize the modeled expected query time. However, the scoring function still needs pre-calculated data structures. This is discussed in depth in the next section.

Secondly, the algorithm does not specify the stopping criteria to determine when improvements are not "significant". This is because multiple criteria can be used. For example, improvements could be considered significant if more than a specified threshold of documents changed clusters in the last iteration. However, the scoring function gives another criteria. If the total sum of scores for each document does not improve by a threshold value compared to the last iteration, the improvement is not considered significant. This enables a direct tradeoff between clustering run times and the desired quality. Because the score is directly linked to the expected query time, the threshold value indicates which improvements are considered significant enough for another clustering round to be worthwhile. This thesis uses a fixed threshold and considers improvements as significant if the total score is less than 99 % of the score from the last iteration. Note that lower scores are better.

Because the number of iterations in each recursion level are indeterministic, an exact time analysis of the whole algorithm is not possible. However, the time complexity of a single iteration of each recursion level can be determined. All documents of a recursion level have to be assigned to K clusters. If scoring occurs in constant time this gives a time complexity of $\mathcal{O}(K \cdot |D|)$ for each iteration. Even though the scoring function presented in this thesis runs in $\mathcal{O}(|d|)$ it can be argued that the average document size is a constant of the document collection and independent from the clustering algorithm.

4.4.2 Document-Grained vs Iteration-Grained

Section 4.3 gives a detailed analysis on how the scoring function used in this thesis works. The fast computation of the scoring function depends heavily on pre-calculated data structures. While this makes the scoring fast, overhead can severely limit the performance of the clustering algorithm. Two alternatives are discussed here. One is *document-grained*,

that is the data structures are updated after each document is added and the other is *iteration-grained*, where the data structures are updated only after all documents where assigned in each iteration.

Accuracy can suffer by only recomputing after all documents have been assigned to clusters. Three observations justify why a loss in accuracy is acceptable:

- The score of a document is primarily determined by the terms that occur in the cluster more frequently than terms in the document. If the score is low, meaning good, most terms in a document occur often within a cluster. By adding a document, occurrences of these terms will remain large. Clusters with low values for a document therefore are unlikely to change the order of terms with regard to occurrences after adding the document.
- Properties of clusters with many documents are unlikely to change greatly after adding a few documents.
- By iteratively reassigning documents, clusters become more stable and loss in accuracy will become less significant.

However, for small clusters this is not necessarily the case. This thesis takes a pragmatic approach. In deeper recursion levels, where clusters are smaller, the data structures are recalculated after each document assignment. This ensures an exact solution, but is only feasible if a fraction of documents are considered. In higher recursion levels, where clusters are larger, data structures are only recalculated after all documents were assigned in each iteration. Since clusters are reasonably stable at this point in the algorithm, the performance gain outweighs the potential loss in accuracy, which is expected to be small.

In practice, clusters with about 50–100 documents seem to be big enough to recompute the data structures only after each iteration. When clusters are much smaller, an interesting effect occurs. With smaller clusters, the number of terms with potentially more occurrences decreases. Because only terms with more occurrences than terms from the document influence the score, this leads to a lower and better score.

Sometimes a single cluster has a very low score for most documents in the collection. This happens when a particular cluster is very small compared to other clusters. Since the data structures for the scoring function is not adapted after every document, most documents are assigned to the single small cluster. In the next iteration, since all documents score very badly for this cluster because it is huge in comparison to other clusters, documents are reassigned to other clusters leaving the originally huge cluster nearly empty.

The result is an oscillation between a huge cluster with nearly all documents and a nearly or even totally empty cluster. However experiments show this effect disappears with sufficiently large clusters or recalculation after each document.

4.4.3 Fast Hierarchical Clustering

The FMClustering algorithm presented in Section 4.4 has a time complexity of $\mathcal{O}(K \cdot |D|)$ per iteration. This can be improved by the observation that clusters tend to be hierarchical by nature. Several hierarchical clustering algorithms have been proposed in the past and successfully applied in many domains [SKK$^+$00].

The FMClustering algorithm is used as a foundation for the recursive Algorithm 3 inspired by the bisecting K-Means algorithm [SKK$^+$00]. The documents are initially clustered into

χ clusters, where χ is small compared to the amount of final clusters K. Each cluster is then recursively clustered again into χ clusters until the minimum size of a cluster is reached.

Algorithm 3 TopDownClustering

Require: Documents D, minimum cluster size s_c, split factor χ, shrink factor SF
1: **if** $|D| \leq s_c$ **then**
2:    **return** D
3: **end if**
4: $C' \leftarrow$FMClustering(D, χ, SF)
5: $C \leftarrow \emptyset$
6: **for all** $c' \in C'$ **do**
7:    $C \leftarrow C \cup$ TopDownClustering(c', s_c, χ, SF)
8: **end for**
9: **return** C

The unknown number of iterations in the FMClustering algorithm makes an exact time analysis difficult again. However, under assumption that the number of iterations in the FMClustering algorithm is a small constant, the TopDown algorithm runs in $\mathcal{O}(|D| \log \frac{|D|}{s_c})$. With the choice of $s_c = \frac{|D|}{K}$ this effectively becomse $\mathcal{O}(|D| \log K)$. This is a nice property, as the number of clusters does not weigh in as a linear factor and makes clustering for many clusters considerably faster.

Apart from being faster, there are several other advantages with a recursive algorithm. For one thing, a hierarchical clustering might be beneficial in some scenarios. For example, using super-clusters consisting of several smaller clusters can be used to distribute clusters among shards, as discussed in Section 5.4. Moreover, recursively splitting up clusters allows more control over the size of clusters. Recursion can be stopped once the desired size is reached.

A more sophisticated stopping criteria also might be used. One example is to use a similarity measure of documents within a cluster instead of the size of a cluster. The expected query time from Equation 4.2 suggests a promising approach: Stop clustering once the ratio between the expected query time and number of remaining documents lies beneath a predefined threshold. However this thesis limits its experiments to the simple size criterion.

4.5 Optimizations

This section looks at practical and pragmatic attempts to increase clustering speed. While some optimizations diverge from the theoretical analysis, experiments show that they work well in practice. The optimizations discussed here are parallelization, reducing the overhead from the pre-calculated data structure and using a faster scoring function.

4.5.1 Parallelization

Parallelization of the TopDown clustering algorithm is straightforward and simple. Iterating over all subclusters $c' \in C'$ can be done in a parallel loop as there are no data dependencies between the runs.

The FMClustering algorithm is not this straightforward but still simple. The focus lies on the loop of assigning documents to clusters. The rest is considered to have negligible impact on performance. Scoring documents can be easily parallelized by calculating the score for each cluster in parallel.

The algorithm itself can be applied to the MapReduce programming model [DG08]. MapReduce has been successfully applied to K-Means [CKL$^+$07]. With the iteration-grained pre-calculation this process is straightforward. In the Map-step each document is scored and the best cluster is determined. In the Reduce-step all documents for a cluster are gathered together and the cluster data structures are recalculated using the new documents.

For document-grained preprocessing parallelization is more complicated. Documents can not be easily added to clusters in parallel, because clusters change with every document added and the scoring might be outdated and wrong. There are two possible solutions to this problem. One option is to simply accept outdated and wrong scores. Since local search clustering algorithms are heuristic, this can be an acceptable solution and wrong or outdated scores may not have a big impact. The other option is to implement a sophisticated locking scheme or similar measures. However this thesis does not go into details.

4.5.2 Term Cutoff

To use the data structures discussed in Section 4.3 a pre-computation step with time complexity of $\mathcal{O}(|T|)$ is necessary. Furthermore, the arrays L_1^c and L_3 use $\mathcal{O}(|T|)$ space. With K clusters this leads to a total time complexity in $\mathcal{O}((K+1)|T|)$ and space complexity in $\mathcal{O}((K+1)|T|)$ for calculating L_3 and L_1^c for all clusters.

However, according to Zipf's law only a few terms account for most occurrences in both queries and documents [BY]. This means $p(t)$ will only be large, and therefore important to the scoring function, for a few terms while most terms will have a small $p(t)$. These terms will be insignificant with regard to the scoring function. Furthermore, the terms with large $p(t)$ are more common in documents, which leads to a consolidation of this effect.

This suggests a simple optimization: Disregard terms with low $p(t)$. Terms are ordered with regard to $p(t)$ and documents are filtered by leaving only the top j terms in the document. With $j \ll |T|$ pre-calculating is reduced to $\mathcal{O}((k+1)j)$ time and space. Let d_j be the document d consisting only of the top j terms. Then scoring a document for a cluster is additionally reduced from $\mathcal{O}(|d|)$ time to $\mathcal{O}(|d_j|)$ time.

4.5.3 Omitting α_2 and α_4

Calculating α_2 and α_4 requires sorting the terms of a document by their frequencies. This is an expensive operation compared to the simple table lookups used for α_1 and α_3. Furthermore α_2 and α_4 only touch a fraction of terms from the document collection and are therefore negligible compared to α_1 and α_3. A simple optimization is to omit α_2 and α_4, rendering the sorting unnecessary and reducing the calculation overhead.

4.6 Summary

This chapter introduces 2 clustering algorithms which can be parallelized and scale to millions of documents. A semantically meaningful scoring function which explicitly minimizes

query times completes the clustering algorithms. An analysis shows that an extremely fast calculation is possibly with pre-calculation, leading to a few simple table lookups per document. The clustering speed can further be improved by several optimization tricks described in Section 4.5.

5. Search with Document Clusters

While the previous chapter focuses on how to cluster documents, this chapter explains in detail how set intersection algorithms can be altered to use a clustering of documents to improve intersection speed. There are two main approaches investigated and discussed here. Firstly, a new data structure which explicitly supports clusters and secondly, a relabeling scheme which improves the performance of existing algorithms. Lastly, a brief overview is given on how clustering can be used in a term- or document-partitioned distributed search engine.

5.1 Set Intersection Data Structure for Clusters

Dividing the documents into clusters requires a new data structure for accessing posting lists. The Lookup algorithm described by Sanders and Transier [ST07] serves as starting point to the data structure and algorithm presented here. Lookup has been shown to work well in theory and practice for real world datasets. The data structure used by the Lookup algorithm is referred to as *Lookup list*. See Section 3.4 for a detailed description for the Lookup algorithm.

Algorithm 4 CLUSTER-LOOKUP(t_1, t_2)

Require: Lookup lists

1: $l_1 \leftarrow$ Lookup list for clusters and term t_1
2: $l_2 \leftarrow$ Lookup list for clusters and term t_2
3: $C \leftarrow \text{lookup}(l_1, l_2)$
4: $D \leftarrow \emptyset$
5: **for** $c \in C$ **do**
6:     $l_1 \leftarrow$ document Lookup list for term t_1 and cluster c
7:     $l_2 \leftarrow$ document Lookup list for term t_2 and cluster c
8:     $D \leftarrow D \cup \text{lookup}(l_1, l_2)$
9: **end for**
10: **return** D

The basic idea is to use a 2-level Lookup algorithm pictured in Algorithm 4. On the top-level, clusters are intersected using the Lookup algorithm on *cluster posting lists*. The

cluster posting list for a term t contains the ID of a cluster if, and only if, the cluster contains at least one document with that term. From this it follows that intersecting the cluster posting lists for a conjunctive query yields all clusters containing at least one document for either term. Note that this is not necessarily the same document for both terms. However, documents containing both terms will only be in clusters returned by the intersection algorithm.

Document posting lists are kept separate for each cluster. A separate Lookup list is therefore needed for each cluster c and every term t that occurs in cluster c. These lists are referred to as *document Lookup lists*. The final set of document IDs are then retrieved by intersecting the document Lookup lists for each cluster found by the cluster intersection. The results for each cluster are joined into the final set.

The data structure containing both the Lookup list for the clusters and the document Lookup lists for each cluster is referred to as *Cluster-Lookup list*. In general, there is a Cluster-Lookup list for each term.

Fast access to the individual Lookup lists is important. For the Lookup lists for clusters, this can easily be achieved using a term-indexed array. This provides optimal access in $\mathcal{O}(1)$ and is a compact representation because a Lookup list is needed for each term t.

For the document Lookup lists however, not every term t might occur in a cluster c. Using a term-indexed array would be a waste of space as entries would be sparse. However there is a one-to-one relationship between an element in the Lookup list for clusters and its corresponding document Lookup list. The position of a cluster c in the Lookup list for term t can be used as an index to an array of document Lookup lists for term t as illustrated in Figure 5.1. This provides easy and fast $\mathcal{O}(1)$ access to the document Lookup lists for every term t. The position of a cluster c within the Lookup list for clusters is known during traversal. An alternative approach is to piggy-back the document Lookup list as part of each element in the Lookup list for clusters.

Since Lookup has an intersection time complexity that lies in $\mathcal{O}(m)$, where m is the size of the smaller set, this data structure achieves the time complexity that is used in the cost function from Equation 4.1 with $\Phi(m, n) := \min(m, n)$.

5.2 Document ID Relabeling

While the data structure presented in Section 5.1 provides an optimal solution with regard to cost function from Equation 4.1, it introduces some considerable overhead due to the level of indirection caused by having multiple document Lookup lists per term. This section discusses an alternative approach without the theoretical guarantees which proves to work well or even better in practice as shown in Section 7.4. It is discussed on the basis of the Lookup algorithm, but can be applied to most set intersection algorithms.

The theoretical analysis of the Lookup algorithm from Transier [Tra10] assumes randomly distributed document IDs. The analysis also mentions that relabeling document IDs in a non-random order might be beneficial to both compression and actual run times, but does not research this hypothesis.

The basic idea is to assign consecutive IDs to documents with documents grouped together by cluster membership. In other words, documents from the same cluster will have consecutive document IDs. For the remainder of this section assume the original document

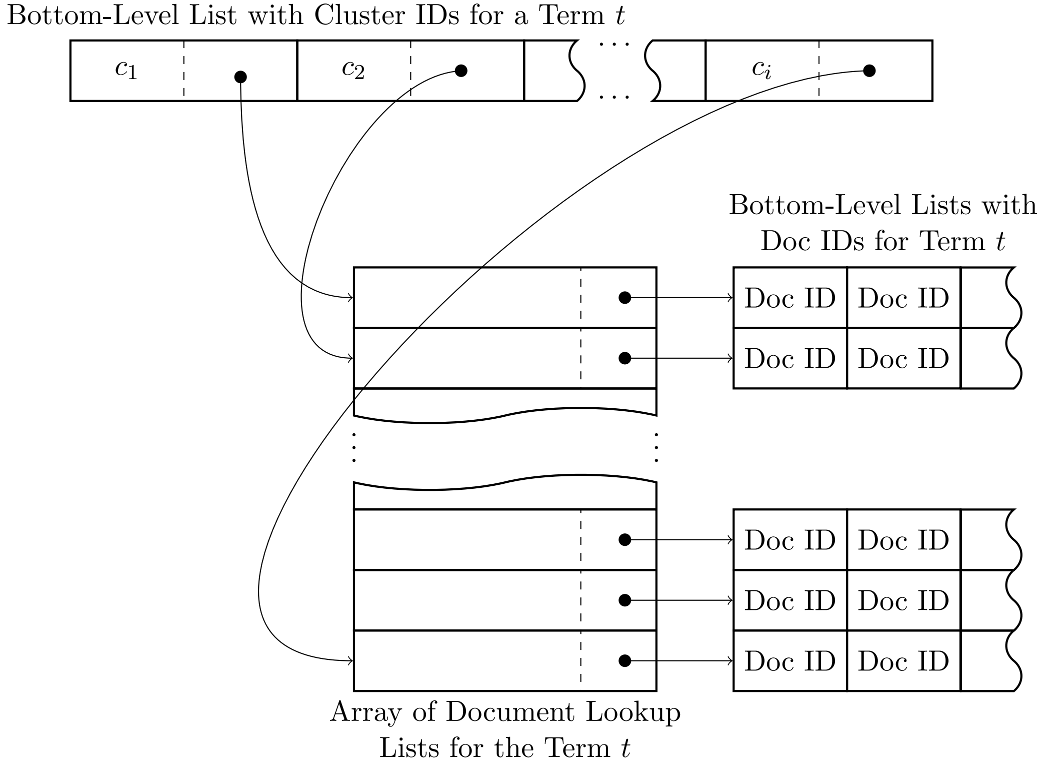

Figure 5.1: The index of a cluster c in the Cluster-Lookup list for a term t is used to resolve the document Lookup list for cluster c and term t

IDs are uniformly distributed in the range from $[0, U]$. Posting lists will have document IDs evenly distributed over the entire range as illustrated in the first chart of Figure 5.2. After clustering and relabeling, document IDs will be clustered together, similar to the second chart. The strength of this effect depends on how well a term t was clustered by the clustering algorithm.

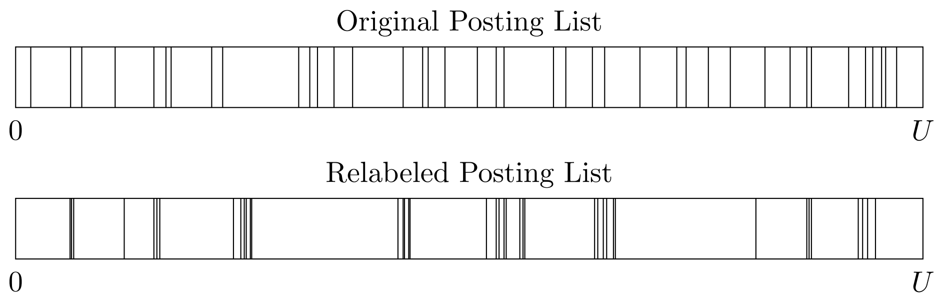

Figure 5.2: The position of the document IDs in the range $[0, U]$ before and after relabeling.

The benefit of relabeling is twofold:

Run Times

Two effects come into play here. For one thing, the scoring function explicitly tries to split up common terms into different clusters. This leads to small overlapping between clusters for two common terms. Through relabeling, documents from a particular cluster can only

fall into a small range of buckets from the Lookup list. Two buckets for two different, common terms will have little or no overlap. For the Lookup algorithm this means that many buckets can be skipped, leading to less buckets visited.

The other effect is that bucket sizes become skewed. While this proves bad for the theoretical analysis, this is not a problem in practice. As Transier shows, entering a new bucket is relatively expensive compared to traversing the bucket. Because of the clustering, multiple hits within buckets are more likely. This has positive influence on caching behavior and branch prediction.

However, the theoretical runtime still is no better than $\mathcal{O}(m)$ as all elements from the smaller set have to be traversed.

Compression

When using compressed posting lists, the document IDs are commonly encoded as differences between IDs, referred to as Δ-encoding. For variable-length encodings, such as Elias-γ coding [Eli75], less bits are used for smaller values. By relabeling the documents, gaps between document IDs decrease, and therefore the average number of bits needed to encode a document ID decreases.

5.3 Hybrid and Parallel Intersections

The Cluster-Lookup algorithm and data structure presented in Section 5.1 come with a notable overhead. For each cluster matched in the intersection, a separate run of the Lookup algorithm has to be carried out. This is involves dereferencing the Lookup data structure and initializing the Lookup algorithm variables. When the number of documents for a term in a cluster is low, this becomes a serious and noticeable overhead.

A hybrid model can keep this overhead low. Instead of keeping a separate Lookup data structure for all clusters, one data structure is used for a single super-cluster consisting of several smaller clusters. Within this super-cluster however, the document IDs are relabeled using the smaller clusters. This combines the best of both worlds. For one thing, the Cluster-Lookup can have an asymptotically better time complexity than the standard Lookup algorithm. Moreover, the algorithm can take advantage of the benefits introduced by the Document ID relabeling scheme.

Using *Cluster-Lookup lists* makes parallelization straightforward. Each cluster can be intersected in parallel as there are no data dependencies between clusters. This makes it a good choice on modern multi-core architectures.

5.4 Distributed Search and Document Clusters

Distributed Search has two common partitioning schemes, by document and by term. Document clusters can be used to improve performance in both partitioning schemes.

Document Partitioned Indexes

Applying document clusters to document partitioned indexes is straightforward. Documents are assigned to shards based on cluster assignment. In the case of hierarchical clustering, document clusters within the same super-cluster are assigned to the same shard.

If the broker keeps an index of which shards contain documents for a particular term, a shard can be omitted for a query if it does not contain any documents for any of the terms in the query. The clustering can help to minimize the amount of shards that contain documents for a term. Communication overhead sinks with less shards. However, load unbalancing might become a problem.

In general the size of the clusters will be smaller than the number of documents assigned to each shard. Multiple clusters per shard can be used to employ the techniques discussed in the previous sections.

Term Partitioned Indexes

In a term partitioned index each shard holds only a subset of all terms. Therefore each document is distributed among multiple shards. This prohibits the use of document clustering directly. However, a separate clustering can be used for each individual shard. Since each shard only holds a subset of terms, the clustering should only take into account the terms actually indexed by that shard. Experiments suggest that using less terms can improve the clustering quality.

However, this is only useful if terms that typically occur together in conjunctive queries reside on the same shard. Luccese et al. [LOPS07] address the problem of assigning terms to shards. They explicitly try to minimize the communication overhead that occurs if terms in a single query are distributed over multiple shards.

6. Implementation

To control every aspect of the indexing stage and to be able to adapt to the needs of this thesis, a small and simple full text search library was developed. The library is written in the lazy functional programming language Haskell[1]. Where execution times are a prime concern the engine uses C libraries and Haskell's foreign function interface to call the library functions from within Haskell. The library includes over 4000 lines of Haskell code and nearly 2000 lines of C code. The implementation is available under the *New BSD Licence* on GitHub[2].

The search engine is split into a core library and separate command line tools for the different processing steps. The core library provides all the basic data types and functionality. The tools make heavy use of the functionality provided by the library.

There are four processing steps:

1. **Tokenization**. A collection of raw documents is processed into a binary representation. Documents are split into words and each word is added to a dictionary and assigned an integer *term ID*. Documents are now represented as an array of *term IDs*. The dictionary, meta information for each document and term frequencies are all saved on disk together with container files holding the term representations of the documents. This uncouples tokenization from later steps and makes using the same document collection for different purposes easy.

2. **Index Construction**. Using the document collection from step one an inverted index is constructed and stored on disk.

3. **Clustering**. The clustering step is optional, but essential to this thesis. The clustering algorithm clusters document collections. Clusterings have a binary representation and can easily be persisted to disk.

4. **Evaluation**. The inverted index can be used to perform several tasks, such as evaluating different measures or running an interactive search engine. Clusterings from the previous step are used where appropriate.

[1] `http://www.haskell.org`
[2] `https://www.github.com/jdimond/diplomarbeit`

6.1 Core Library

The core library provides all the necessary data types and functions that are needed to implement a search engine. The core library is divided into several modules with distinct functionality. The functionality includes, but is not limited to, text processing functions, search support, reading and writing of core data structures to disk and clustering algorithms.

6.1.1 Basic Data Types

The core library makes extensive use of Haskell's type system to ensure correctness. It provides the basic data types for things like documents, terms and inverted indexes. Everything is strongly typed and the library uses Haskell's `newtype` mechanism to ensure type safety without performance penalties at runtime whenever possible.

Types defined with `newtype` use another type as runtime representation. However, types are not interchangeable, even if their runtime type is the same. For example, `DocId`, `TermId` and `ClusterId` are defined as unsigned 32-bit integers. At runtime there is no performance penalty compared to using simple unsigned integers. However, the compiler checks the type and a `DocId` can not be used where a `TermId` is expected. This reduces the number of mistakes that can occur by using the wrong kind of IDs.

The same applies to other data types. Documents are represented as an array of unsigned integers at runtime, but as lists of `TermId`s at compile time.

6.1.2 Text Processing

Documents are parsed from the input format and loaded as UTF-8 strings into memory. Documents are then split into terms using the token separators from Table 6.1. The table is taken from Transier [Tra10] with small modifications. Terms are converted to lower-case and filtered using the stop word list in Figure 6.1, which consists of the most common English words[3].

		!	”	#	$	%	&	’	(
<0x20	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28
)	*	+	,	-	.	/	:	;	<
0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f	0x3a	0x3b	0x3c
=	>	?	@	[	\	]	ˆ	_	‘
0x3d	0x3e	0x3f	0x40	0x5b	0x5c	0x5d	0x5e	0x5f	0x60
{	|	}	˜						
0x7b	0x7c	0x7d	0x7e	0xa0	0xb0				

Table 6.1: Token Separators with Unicode Code Points

6.1.3 Clustering

The clustering algorithms described in Section 4.4 are implemented in C to ensure maximal performance and competitive execution times. Before clustering starts, documents are loaded into main memory to ensure disk I/O has no influence on the runtimes of the

[3] The list was taken from `http://www.textfixer.com/resources/common-english-words.txt`

a, able, about, across, after, all, almost, also, am, among, an, and, any, are, as, at, be, because, been, but, by, can, cannot, could, dear, did, do, does, either, else, ever, every, for, from, get, got, had, has, have, he, her, hers, him, his, how, however, i, if, in, into, it, its, just, least, let, like, likely, may, me, might, most, must, my, neither, no, nor, not, of, off, often, on, only, or, other, our, own, rather, say, says, should, since, so, some, than, that, is, what, she, said, the, their, them, then, there, these, they, this, to, too, us, wants, was, we, were, when, where, which, while, who, whom, why, will, with, would, yet, you, your

Figure 6.1: Stop Word List

algorithm. More sophisticated implementations will probably want to implement some kind of document caching to fine tune the tradeoff between memory footprint and algorithm performance.

The C implementation uses OpenMP[4] to support parallel execution.

Implementation Details

Documents are represented as `uint32_t` arrays. Documents are all loaded into memory prior to clustering. Terms are filtered according to the chosen Term-Cutoff value and term IDs are reassigned to a continuous range. Duplicate terms are removed while loading. This improves the efficiency of the scoring function as this step becomes unnecessary during the actual scoring.

Instead of randomly sub-sampling the documents in each recursion, the order of the documents is randomized once in the beginning of the algorithm and the document IDs stored in an array. Randomly sub-sampling the documents can then be achieved by simply taking the first elements from this array which runs in constant time. The same array can be used for all recursion levels.

In order to calculate the lookup table L_1^c each cluster holds an array counting the term occurrences for each term in cluster c. Additionally there is a global array counting in how many clusters a term t occurs. This is used to calculate the lookup table L_3. All these arrays, including the lookup tables are indexed by term.

If the amount of documents is smaller than than $k \cdot 100$, where k is the number of clusters, document-grained recalculation of the lookup tables is used. In all other cases iteration-grained recalculation is used.

Document-grained recalculation is implemented without support for parallel execution. This simplifies the code considerably. The lookup tables L_1^c and L_3 are only recalculated if the assignment of a document to a cluster changes. This results in a huge speedup if documents stay in the same cluster in between iterations.

For iteration-grained recalculation, documents are first assigned to the clusters. This is done in parallel using the OpenMP `omp parallel` pragma and dividing the documents among the threads. The new term frequencies for each cluster are calculated in parallel by dividing the clusters among threads. Finally, the lookup tables are calculated in the same way.

[4] http://www.openmp.org

There are two implementations for the scoring function which can be chosen at runtime. The first calculates the inexact scoring function by omitting α_2 and α_4 as described in Section 4.5.3. The second calculates the scoring function like in the original definition from Theorem 4.3.4. However, sorting of the terms is done in $\mathcal{O}(n \log n)$ with the standard C `qsort` function.

For the TopDown algorithm the minimum size per cluster is determined by $s_c = \frac{|D|}{k}$ where k is the desired amount of clusters. The split factor is set to $\chi = \min(8, \frac{|D'|}{s_c})$, where $|D'|$ is the number of documents in the current recursion level. Because the TopDown algorithm recursively splits clusters until a minimum size is reached, the final amount of clusters can vary between runs. To circumvent this, a post-processing step is applied when using this algorithm. If the final amount of clusters is larger than the desired amount of clusters, the two currently smallest clusters are merged until the desired amount of clusters is reached. In cases where there are less clusters than desired, the currently largest cluster is split into two equally sized clusters until the right amount of clusters is reached. Documents are randomly distributed among the two new clusters. While this does not improve the quality of the clustering it makes the number of clusters deterministic and comparable.

In deep recursion levels it can happen that the TopDown algorithm produces tiny clusters. This can happen if the documents are very similar and a few outliers form a separate cluster. To prevent tiny clusters, clusters are merged together if the size lies below a certain threshold. The threshold is set to half the minimum size s_c of a cluster.

6.1.4 Set Intersections

The Lookup algorithm [ST07] is implemented in pure C to achieve maximal performance. The algorithm is accessed via Haskell's Foreign Function Interface. The overhead of calling native C code from Haskell has similar overhead to a normal C function call when the `unsafe` keyword is used.

The Cluster-Lookup algorithm from Section 5.1 is implemented in pure C as well. The Lookup algorithm used to intersect clusters and documents is based on the basic Lookup algorithm to ensure comparable results.

The bucket size is set to $B = 16$ for document Lookup lists and $B = 8$ for cluster Lookup lists. This seems a good tradeoff between space and speed requirements.

6.2 File Formats

Each processing step saves its intermediate result on disk to enable repeatable experiments in the next processing step. This section gives a small overview over the file formats used. When not specified otherwise, the binary representations of integers are in the big endian format. Term IDs, document IDs and cluster IDs are all represented by unsigned 32-bit integers.

Document Collection

The document collection is a folder containing multiple files. Documents are saved in a container format to keep the number of files low. An index file holds all the necessary information to access the documents.

Index File This file contains an array of fixed width entries. The entry at position i corresponds to the document with ID i. An entry consists of an 8-bit unsigned integer which specifies in which container the document is stored. It is followed by a 32-bit unsigned offset within the container and a 16-bit length of the document. The length is given in bytes. Thus a document is limited to $\frac{65536}{w}$ terms, where w denotes the size of a term in bytes. In this case $w = 4$.

Container File A container file holds concatenated documents as array of terms. A term is a 32-bit unsigned integer. Container files are limited to 4 GiB in size due to the size of the offset in the index file.

Term Frequencies The term frequencies are stored as an array of 32-bit unsigned integers. The position within the array corresponds to the term ID.

Meta File Each document has a meta record associated with it. The meta record is of arbitrary length. The meta records are stored in this file sequentially without any separator tokens.

Meta Offsets The meta offset file holds an array of 64-bit unsigned integers. The integer at position i represents the offset of the meta information for the document with ID i in the meta file.

Dictionary The dictionary consists of concatenated null-terminated UTF-8 strings. The position in the dictionary is the corresponding term ID. This implies that the dictionary can not be loaded partially.

This design has several limitations. However, the design choices are a tradeoff between necessary functionality, simplicity and access times. For example, the length of documents is limited to 16 384 terms and the size of the processed collection to 1 TiB. Larger documents are truncated. Limits can easily be raised by using larger integers. However, the file formats for the document collection are easy to read and write and suffice for experiments in this thesis. The design also supports easy, fast and memory efficient loading of single documents.

Clustering

A clustering is in principle a mapping from cluster IDs to sets of document IDs. The clustering is saved to disk as follows. A 32-bit unsigned integer n specifies the range of mappings. A mapping must exist for IDs 0 until $n - 1$ inclusive. This is followed by a 32-bit unsigned integer m which specifies the number of non-empty mappings. A mapping is considered non-empty if it maps to a non-empty set.

This is followed by m entries. Each entry stores a 32-bit unsigned integer which is the original cluster ID in the mapping followed by a 32-bit unsigned integer which represents the size of the set. Finally, in order, the set for each entry is stored as an array of 32-bit unsigned integers representing the document IDs.

Inverted Index

The inverted index is essentially a mapping from term IDs to sets of document IDs. The same file format as described for the clustering is used. The only difference is that term IDs are used instead of cluster IDs and document IDs are sorted within sets. By loading the array of entries into memory, individual posting lists can be loaded into memory on demand.

6.3 Tools

Several command line tools take care of the processing steps. The most important tools are described here.

6.3.1 Tokenization

The tool `Tokenize` converts an input of raw documents to the format for document collections specified in Section 6.2. It allows multiple input formats, including documents as plain UTF-8 text files, HTML files and container formats. Container formats include the GOV2[5] file format and concatenated, length delimited UTF-8 texts in plain and compressed formats. The tool also supports multi-threading. However, synchronized access to the dictionary and single threaded writing to the document collection limits the speedup. The implementation uses streaming techniques to keep the memory footprint low. Documents are converted into lists of terms and then saved to disk immediately. Only the dictionary and term frequency counters are kept in main memory during the entire process.

6.3.2 Index Construction

Using the document collection, the tool `BuildIndex` can construct the inverted index described in Section 6.2. Special care is given to the memory footprint of index construction. The document collection is split into subsets and the inverted index is constructed separately for each subset. This requires less memory. In a final step all inverted indexes are joined. This can happen with a relatively small memory overhead as each posting list can be handled separately.

The tool also supports concurrent construction of the index. However, because random disk accesses are used excessively, concurrent construction can descrease the performance when random disks access is slow, as is the case with magnetic hard drives.

6.3.3 Clustering

The tool `Cluster` performs the clustering illustrated in Section 4.4. The tool is written in Haskell but uses a C implementation of the algorithms to ensure fast and efficient clustering, as described in Section 6.1.3.

The command line parameters control the different aspects of the clustering algorithm. The tool can use either the fast scoring method or the exact scoring method. This exact scoring function is used if the `-exact` flag is specified. The tool can also cluster subsets of the document collection if specified.

6.3.4 Evaluation

There are different tools for the evaluation stage. Here is a short overview over the different tools.

[5]See Section 7.3.1 for a detailed explanation on GOV2

Benchmark

The `Benchmark` program takes a query log, a document collection and a clustering and evaluates different measures. This includes the theoretical speedup that can be achieved with this clustering[6]. The tool also measures the actual speedup between the naive Lookup algorithm and the variants described in this thesis. The tool uses the `criterion`[7] library to measure the intersection times for queries. The number of runs can be given as a command line parameter. The statistics are printed to the standard output.

Search

This tool provides an interactive search for conjunctive queries on the command line. It uses the meta data generated in the first step to load the first n documents for a query and presents them on the command line. It also displays the total number of documents matching a query. The documents are not ranked.

Stats

The `Stat` tool prints several interesting statistics from a document collection to the standard output.

Compression

Compression ratios of the posting lists can be determined with the `Compression` tool. For each term, it outputs the number of elements in the corresponding posting list together with the size of the posting list for several encodings with and without relabeling.

[6] This is explained in depth in Section 7.1
[7] version 0.8.0, `http://hackage.haskell.org/package/criterion`

7. Experimental Evaluation

This chapter evaluates the algorithms introduced in the previous sections on real world datasets. The experiments are chosen to give insights into the most valuable configurations and parameters. The large amount of data prohibits an evaluation of all combinations of parameters. However, important tendencies and observations are investigated using carefully chosen representative examples.

7.1 Evaluation Measures

The quality of the clustering algorithms is evaluated using three different measures. The first is the theoretical speedup based on the theoretical time complexities for set intersections. The other two are real speedups based on measuring actual execution times for set intersections.

Theoretical Speedup

Equation 4.1 defines the cost function $\Psi_C(t_i, t_j)$ for a query $q = \{t_i, t_j\}$ as a function of a cost function $\Phi(n, m)$ for intersecting two sets of size n and m. Let $n(t)$ be the size of the posting list for term t using all documents. Given a testing query log Υ and a clustering C the cost function Ψ_C is used to calculate the *theoretical speedup* S_T, which is defined as

$$S_T = \frac{\sum_{\{t_i,t_j\}\in\Upsilon} \Phi(n(t_i), n(t_j))}{\sum_{\{t_i,t_j\}\in\Upsilon} \Psi_C(t_i, t_j)} \tag{7.1}$$

The cost function $\Phi(n, m)$ for set intersections is used here as base case without clustering. This is divided by the cost for intersecting posting lists with clustering which is modeled by $\Psi_C(t_i, t_j)$.

Since this thesis focuses primarily on intersections in $\mathcal{O}(m)$, the cost function $\Phi(n, m) :=$ $\min(n, m)$ is set to the minimum of both set sizes for the remainder of this chapter.

Real Speedup

While the theoretical speedup gives an implementation-independent insight into the quality of a clustering, the speed of actual implementations is influenced by many external parameters, e. g., processor speed, implementation details and the compiler that is used. Instead of measuring actual intersection times, we therefore use the speedup of an improved algorithm A in comparison to a base case.

Given measurements for the execution times $T_B(q)$ for each query $q \in \Upsilon$ for the base case and the execution times $T_A(q)$ for an algorithm A the speedup is given as

$$S = \frac{\sum_{q \in \Upsilon} T_B(q)}{\sum_{q \in \Upsilon} T_A(q)} \tag{7.2}$$

The evaluation uses two algorithms for A. The first uses the Cluster-Lookup algorithm from Section 5.1. The speedup is referred to as *cluster speedup* or S_C. The second uses the standard Lookup algorithm with the relabeling scheme discussed in Section 5.2 and is referred to as *relabeling speedup* or S_R. The base case is a standard implementation of the Lookup algorithm.

The algorithms for the base case, the relabeled scheme and the document intersections for the Cluster-Lookup algorithm all use the same implementation. This ensures comparable results and keeps the influence of implementation details at a minimum.

7.2 Experimental Methology

Each experiment consists of multiple runs with varying configurations. A configuration includes the type of algorithm and the parameters used in the algorithm. Each run consists of two parts. First a clustering is generated using the algorithm and the parameters from the configuration. With that clustering, the speedups are benchmarked using a query log. For the real speedups S_C and S_R the benchmark is run 5 times and the arithmetic mean of the total speedups is used as final result.

Each dataset is accompanied by a matching query log. The query log is divided into a training set to learn the language model and a testing set to evaluate the queries. In general only two term queries from the query log are used for testing, while all queries are used to learn the language model.

Because datasets are very large in general, clustering and benchmarking can take a considerable amount of time. This prohibits multiple runs for the same configuration. Exceptions are the experiments from Section 7.4.7 which discuss the stability of the clustering algorithms. However, as Section 7.4.7 shows, this does not pose a serious problem since the results are very stable in general.

The FMClustering algorithm generally takes more time than the TopDown algorithm[1]. Running the FMClustering algorithm for multiple configurations is therefore very time consuming. Even though the FMClustering yields better results than the TopDown algorithm, as shown in Section 7.4.2, the TopDown algorithm is used for most experiments. This is because an extensive evaluation using the FMClustering algorithm is considered too time consuming.

[1] see Section 7.5.1

7.2.1 Environment

Experiments are all conducted on a machine with two octa-core Intel Sandy Bridge Xeon E5-2670 processors with a clock rate of 2.6 GHz. The machine has 64 GiB of main memory. Each processor has a 20 MiB L3 cache, a 256 KiB L2 cache and a 64 KiB L1 cache. The machine runs on SuSE Linux Enterprise Server 11 (kernel version 3.0.42).

The source code is built with GHC 7.6.2 and GCC 4.7.2 with the `-O3` flag enabled.

The process for benchmarking the speedups is pinned to a single processor with 32 GiB RAM to circumvent any influences of non-uniform memory access.

7.3 Datasets

Three independent document collections are used for evaluation. The GOV2s dataset is generated from the GOV2 dataset. An overview of the datasets is given in Table 7.1.

	GOV2	GOV2s	Wikipedia	pagenstecher.de
Documents	25 205 179	631 975 969	6 096 279	786 474
Terms	38 562 580	25 221 691	12 295 297	573 725
Terms / Document	652.22	18.19	230.54	35.70
Input size (raw)	396.74 GB	396.74 GiB	12.37 GiB	175.14 MiB
Inverted Index size	16.25 GiB	32.83 GiB	3.09 GiB	89.8 MiB

Table 7.1: Dataset Statistics

7.3.1 Document Collections

GOV2

The GOV2 test collection [CSC04] is a large portion of available pages from the `.gov` top-level domain in early 2004. The collection comprises of 396 GiB of raw data and contains approximately 25 million documents. Each document is stripped of all HTML tags and converted into plain text before it is indexed.

GOV2s

The GOV2s test collection is based on the GOV2 collection. Documents are split into sentences and each sentence is treated as separate document. The main textual content is extracted from each website using the boilerpipe library[2] based on the paper by Kohlschütter et al. [KFN10]. Sentences are then extracted using the Stanford NLP parser[3].

Wikipedia

Articles from the English Wikipedia are the base for this document collection. Using a dump of all articles from May 2013[4] the articles are converted into plain text using the gwtwiki library[5]. The class `PlainTextConverter` is modified to strip out math and other irrelevant tags. Additional whitespace is added after certain tags to prevent falsely concatenated words.

[2] version 1.2.0, `http://code.google.com/p/boilerpipe/`
[3] version 1.3.5, `http://nlp.stanford.edu/software/corenlp.shtml`
[4] `http://dumps.wikimedia.org/enwiki/20130503/enwiki-20130503-pages-articles.xml.bz2`
[5] version 3.0.19, `http://code.google.com/p/gwtwiki/`

pagenstecher.de

pagenstecher.de is one of the biggest German online communities for car tuning. We use the publicly accessible user posts from it's message board as documents. Each post is treated as single document. This dataset is interesting in two ways. First, the documents concentrate on a single, specialized topic. Second, this dataset comes with a query log of real searches by users of the community. This allows testing of the clustering algorithms on a combination of realistic documents and query logs.

7.3.2 Query Logs

Each document collection is accompanied by a query log. These are partly real world user queries and partly generated artificial query logs. The query logs are split into a training set and testing set. While the sets are in principle disjoint, queries might appear multiple times in a query log. This can lead to overlapping queries between the training and the testing set. However, this is also to be expected with realistic inputs, as queries repeat over time. The terms in queries follow Zipf's law as is shown in Figure 7.1.

	AOL	Wikipedia	pagenstecher.de
Queries	29 077 553	11 000 000	13 230
Distinct Terms	1 501 946	1 067 091	981

Table 7.2: Query Log Statistics

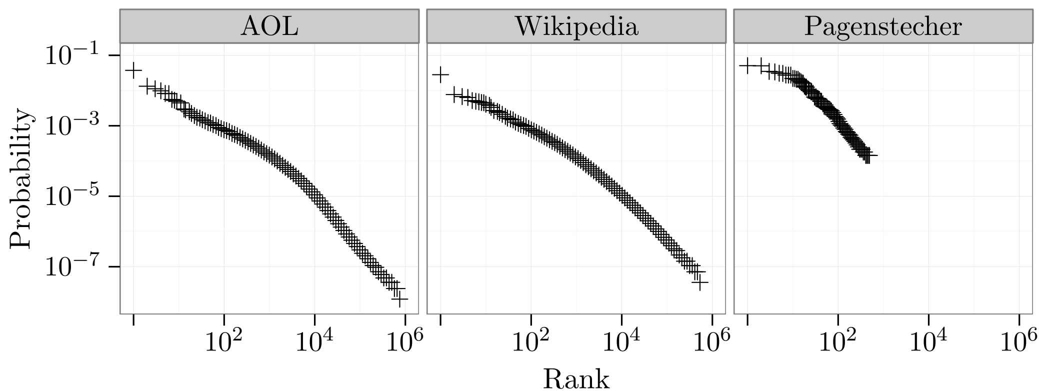

Figure 7.1: Probability of a term appearing in a query as a function of its rank on a log-log scale. A sample of 100 terms with exponentially growing ranks is plotted.

AOL

In 2006 AOL [PCT06] released a collection of 29 million queries collected from about 650 000 users over the timespan of three months. Due to privacy concerns the original data was taken offline shortly after. However, the data is still available on the Internet[6]. We use the logs as semi-realistic input for our GOV2 and GOV2s test collections. Any queries containing a domain are filtered out, as the user probably intended to visit that website rather than perform an actual web search.

[6] `http://www.gregsadetsky.com/aol-data/`

Artificial Wikipedia Logs

Because realistic queries are not freely available for the Wikipedia dataset[7], a simple approach is used to generate artificial queries. All articles are scanned for links to other Wikipedia articles. For each link, the title of the article the link points to is added to the artifical query log. We consider article titles a somewhat realistic input for conjunctive queries and a good compromise for the unavailable real query logs. Furthermore, important articles, i. e., articles with many links pointing to it, are more common in the query log, yielding a query log with term probabilities that follow Zipfs law. This can be seen in Figure 7.1.

Wikipedia titles are generally longer than 2 terms. For example, two of the most common words in the artificial query log, *United* and *States*, in general appear together with other words. This yields mediocre results, because the query language model does not fit the two term queries very well. To circumvent this, the Wikipedia query logs are used differently. For queries that are longer than two terms, all combinations of two terms from that query are used in the evaluation run.

pagenstecher.de Query Logs

The query log for the pagenstecher.de community consists of 13 230 queries entered by real users using the full text search feature on the website. The queries are very specific to the dataset. For example 7 out of the 20 most common search query words are car makes.

7.4 Experimental Results

This section examines the speedups achieved with document clustering. Symbols referring to common parameters and variables are listed in Table 7.3.

Description	**Symbol**
Theoretical Speedup	S_T
Speedup with Relabeling	S_R
Speedup with Cluster-Lookup	S_C
Number of Clusters	K
Number of Documents to Cluster	$\|D\|$
Shrink-Factor	SF
Term-Cutoff	TC

Table 7.3: Algorithm and Evaluation symbols

7.4.1 Speedups

Perhaps the most interesting question is what kind of speedups are possible with document clustering. Plotted in Figure 7.2 are the speedups S_T, S_R and S_C for the three datasets GOV2, Wikipedia and pagenstecher.de as a function of the number of clusters. The clusterings are produced with the FMClustering algorithm with a shrink factor $SF = 0.1$.

[7] see https://blog.wikimedia.org/2012/09/19/what-are-readers-looking-for-wikipedia-search-data-now-available/

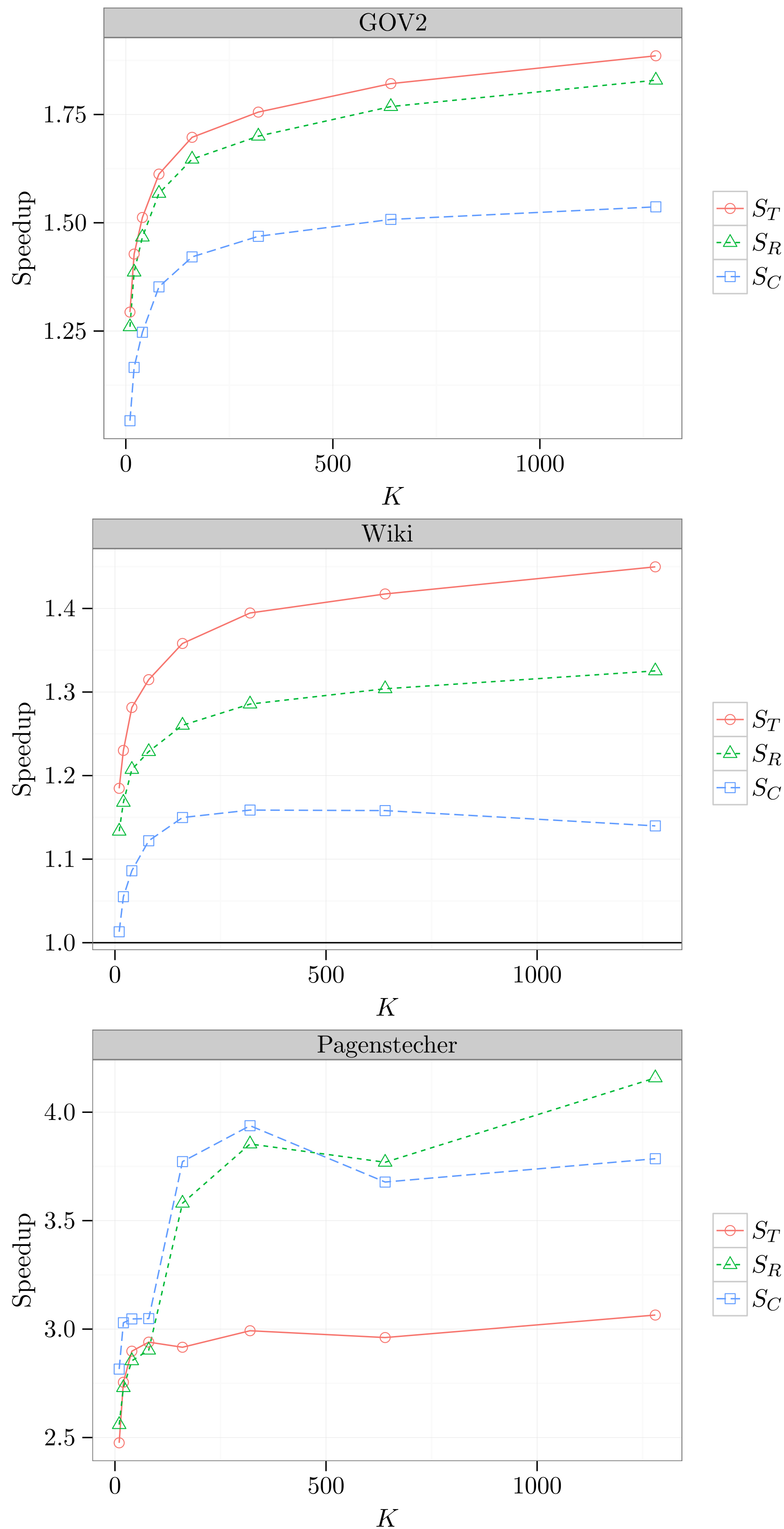

Figure 7.2: Speedups as a function of the number of clusters on different datasets. The FMClustering algorithm is used with $SF = 0.1$ and $TC = 100\,000$.

The speedups depend heavily on the dataset. For GOV2, the best theoretical speedup is $S_T = 1.89$, with the relabeled speedup closely following at $S_R = 1.83$. The speedup with the Cluster-Lookup algorithm is $S_C = 1.53$. In practice, posting list intersections can be done nearly twice as fast with document clustering than without. Wikipedia has a best theoretical speedup of $S_T = 1.45$ and real speedups of $S_R = 1.33$ and $S_C = 1.16$. While this is not as good as with GOV2 there is still a considerable speedup compared to the base case without clustering. Finally, the pagenstecher.de shows the best speedups with $S_T = 3.06$. This is outperformed by the real speedups $S_R = 4.16$ and $S_C = 3.94$. This means that intersecting posting lists occurs 4 times faster on the pagenstecher.de dataset with clustering than without.

All datasets show an increasing speedup with the number of clusters for all three measures. However, the slope quickly decreases with increasing number of clusters. For a large number of clusters, doubling the number of clusters only improves the speedup marginally. The irregularities on the pagenstecher.de set are discussed in detail later in this section.

For the GOV2 and the Wikipedia dataset the theoretical speedup outperforms both real speedups, i. e., S_R and S_C. However, the relabeling speedup S_R is generally higher than the cluster speedup S_C. The lower values for S_C can be explained with the overhead introduced by the Cluster-Lookup algorithm. Even though the algorithm has the time complexity that is used in the theoretical speedup, the constant factors play an important role here. While the theoretical speedup assumes that intersecting the cluster posting lists is as expensive as intersecting the document posting lists this is not true in practice. For each cluster found in the cluster posting list intersection, a new run of the Lookup algorithm has to be bootstrapped. While this runs in constant time, it produces considerable overhead which is noticeable in the lower speedup S_C. Furthermore, the complexity of the Cluster-Lookup algorithm has negative effects on branch prediction and cache efficiency. The overhead also has an influence on the decreasing values for S_C for larger number of clusters in the Wikipedia dataset.

This overhead does not pose a problem for the relabeling speedup S_R. The speedups from relabeling occur because of better branch prediction and cache effects. Branching into a bucket is comparably expensive as it has to lookup the new position in the posting list and initialize several variables. With relabeling, the number of buckets which have to be visited decreases, which improves performance. Buckets contain more elements which leads to better performance in the inner loop intersecting the bottom-level data structure. Even though the time complexity still remains in $\mathcal{O}(m)$, these effects make relabeling achieve speedups that are similar to the theoretical speedup S_T.

The speedups S_T, S_R and S_C show a high correlation in all three datasets. Apart from the jump of the real speedups S_R and S_C in the pagenstecher.de dataset, the theoretical speedup S_T gives a good indication of how good S_R and S_C are. If an empirical evaluation of the real speedup is not feasible, the theoretical speedup can be used to evaluate the best parameters for the clustering.

The plot for the pagenstecher.de dataset shows some irregularities. With $K = 160$ clusters there is a sudden increase for both, the relabeled speedup S_R and the cluster speedup S_C. The theoretical speedup however remains roughly constant for a large number of clusters. Because the inverted index for the pagenstecher.de dataset is small, some portions of the posting lists can remain in the cache in between queries. For common combinations of terms in queries, only a small subset of clusters are relevant. The parts of the posting lists that correspond to these clusters remain in the cache. For a smaller amount of clusters in

the clustering, the (parts of the) posting lists that are relevant become too large and are replaced in the cache in between queries.

Figure 7.3 shows the speedups for the GOV2s dataset using the first 100 000 000 documents of the dataset. This corresponds to the first 100 000 000 sentences of the GOV2 dataset. For reasons of feasibility the TopDown algorithm is used with $TC = 100\,000$ and $SF = 0.1$.

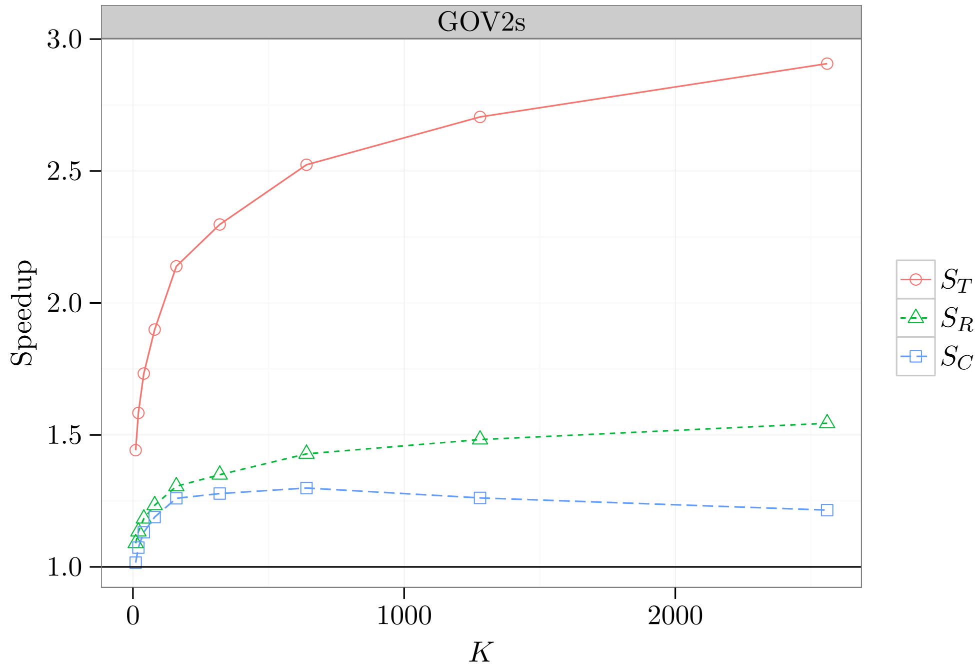

Figure 7.3: Speedups as a function of the number of clusters on the first 100 000 000 documents of the GOV2s dataset. The TopDown algorithm is used with $SF = 0.1$ and $TC = 100\,000$.

The theoretical speedup S_T is much higher on the GOV2s dataset than on the GOV2 dataset for all cluster sizes K. The reason for this lies in the much shorter documents. If documents are shorter, there are less terms that determine which cluster is best. With less terms, there are potentially less conflicts in the scoring function between which cluster is scored best for each term. With $K = 2560$ the theoretical speedup is as high as $S_T = 2.9$. The real speedups S_R and S_C, however, fail to achieve such good values and are generally much lower than S_T. The relabeling speedup S_R, in contrast to S_T and S_C, does not benefit from a better theoretical speedup. This explains why relabeling falls short of the speedups predicted by S_T. For the cluster speedup S_C the discrepancies come from the overhead introduced by the algorithm. On average, the size of the posting list of each cluster is only 40. Considering the impact of branch prediction and cache effects on the runtime, the Cluster-Lookup algorithm struggles to capitalize on the better theoretical time complexities with so few elements. A better implementation could help the actual cluster speedup S_C on the GOV2s dataset.

7.4.2 FMClustering versus TopDown

The speedups on the GOV2 datasets for the FMClustering and the TopDown algorithm are plotted in Figure 7.4. The clusterings are produced with varying cluster sizes K and a shrink factor of $SF = 0.1$ and Term-Cutoff of $TC = 100\,000$. The speedups for

FMClustering are plotted using contoured shapes, the speedups for TopDown use solid shapes. Due to time constraints and the large runtimes of the FMClustering algorithm, the speedups for the FMClustering algorithm are only plotted up to $K = 1280$.

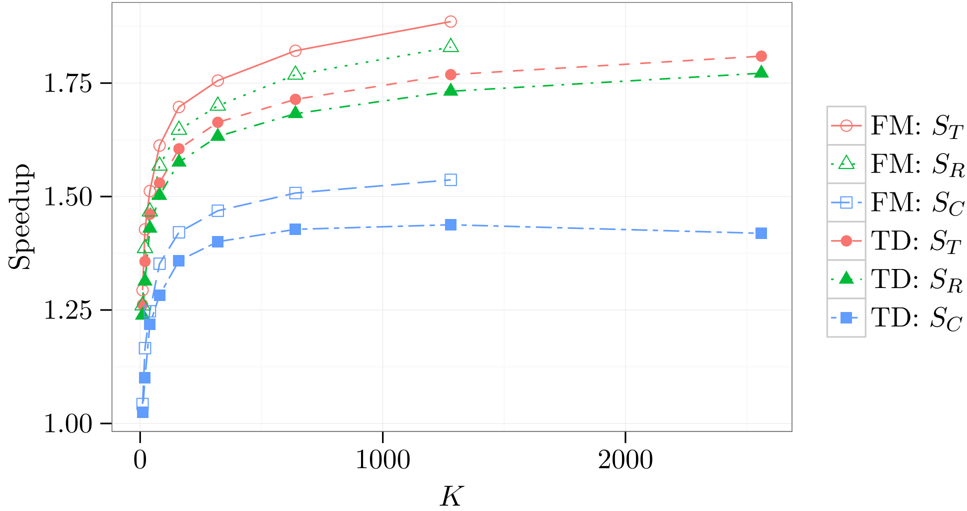

Figure 7.4: Speedups for varying cluster sizes K for the two algorithms FMClustering (FM) and TopDown (TD) on the GOV2 dataset. The shrink factor is set to $SF = 0.1$. The speedups for the FMClustering algorithm are plotted with contoured shapes, the speedups for the TopDown algorithm with solid shapes.

The FMClustering algorithm produces consistently better values than the TopDown algorithm. However, the difference between the individual speedups is small in general. For example for $K = 1280$, the speedup S_T with the TopDown algorithm is only 6.2 % worse than with the FMFClustering algorithm. For the speedups S_R and S_C, the TopDown algorithm is 5.3 % and 6.4 % worse, respectively. Considering the significantly faster run times for the TopDown clustering algorithm, the worse speedups might be an acceptable trade-off. Further, with the TopDown algorithm, a larger amount of clusters can be clustered in equal or less running time, which in general yields better results.

The better results for the FMClustering algorithm compared to the TopDown algorithm can be explained with initially less clusters in the TopDown algorithm. Assume the FMClustering algorithm clusters an arbitrary but fixed subset of documents into 3 clusters. Because the TopDown algorithm initially has less clusters, the same documents might be clustered into only 2 clusters by the TopDown algorithm. Assume documents from cluster 3 are then divided among these two clusters. In deeper recursion levels there is no possibility to form a third cluster from the documents that were split among the clusters in the higher recursion levels because the TopDown algorithm runs independently for all clusters from the current recursion level. This prevents documents from being assigned to better fitting clusters from other recursive runs in the TopDown algorithm. A solution might be to run a final assignment round on all clusters after the TopDown algorithm has finished. However, this is not researched in this thesis.

Effects similar to those described in Section 7.4.1 occur in the TopDown algorithm. The three speedups S_T, S_R and S_C show a high correlation, with S_T being slightly better than

S_R and significantly better than S_C. The speedup S_C decreases for $K = 2560$, while the theoretical speedup still increases. The explanation lies once again in the overhead introduced by the Cluster-Lookup algorithm.

7.4.3 Clustering Effect

For high speedups, terms should appear in as few clusters as possible. This circumstance is illustrated in Figure 7.5. The plot shows the average cumulative distributions of documents in clusters for the 100 000 most common terms split by total posting list length l. The plot shows the percentage of the largest clusters for a term covering the percentage of the documents containing that term. The plot averages over the most common terms. The FMClustering algorithm is used with a shrink factor of $SF = 0.1$, a Term-Cutoff of $TC = 100\,000$ and $K = 1280$ clusters.

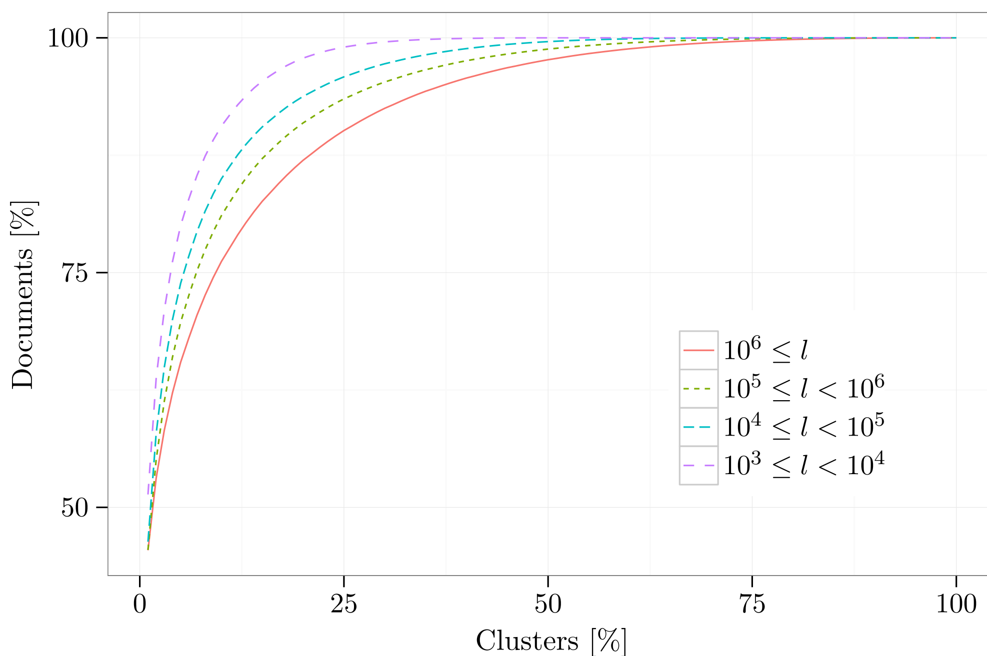

Figure 7.5: Distribution of documents among clusters for different posting list lengths l on the GOV2 dataset for $K = 1280$ clusters using the FMClustering algorithm.

The plot shows that most documents for a term are distributed among only a few clusters. For example, on average 12 clusters, i. e., 1 %, contain roughly 50 % of the documents for a particular term. 50 % of the clusters are enough to cover 97.5 % of documents on average. This increases to 99.5 % if the total number of documents containing a term lies beneath 100 000.

Not surprisingly the clustering works better for terms occurring in less documents. This is shown by faster climbing curves in the plot. Nonetheless, even for terms occurring in more than a million documents—for the GOV2 dataset this corresponds to more than 4 % of the corpus—a heavy clustering effect is still visible. With only 25 % of the clusters 90 % of the documents are covered. When recall is not important, inexact search methods could use this fact to prune a large portion of the clusters. However, for conjunctive queries, documents containing all terms may likely be in clusters that only contain a few documents

for a term from the query. If the clusters are pruned by posting list size, these documents will not be found.

7.4.4 Compression

As discussed in previous sections, the clustering effect can improve compression of the posting lists. Figure 7.6 shows the average number of bits per element for different clustering algorithms and encodings on the GOV2 dataset. The plot is divided into two parts, where one shows the average number of bits per element for the complete index and the other only for posting lists with more than $l > 10^6$ elements.

The compression is achieved using Δ-encoding and different variable-length encodings as discussed in Section 3.5. In the case of the FMClustering and TopDown algorithm, documents from the same cluster are assigned consecutive IDs, as discussed in Section 5.2. $K = 1280$ clusters are used. This is compared to the base case were no ID relabeling is performed. The parameter for Golomb coding is set to $b = \frac{N}{l} \ln 2$ for each posting list, where N is the sum of all differences in the posting list and l is the length of the posting list.

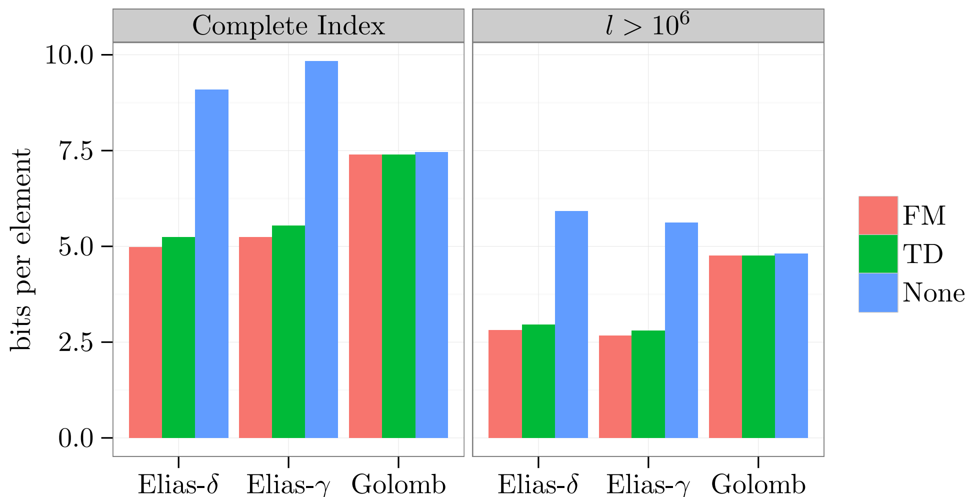

Figure 7.6: Compression of the inverted index using different clustering algorithms and encodings on the GOV2 dataset with $K = 1280$ clusters.

For the base case without relabeling Golomb coding yields the best results. Elias-δ and Elias-γ are much worse. With relabeling however, the opposite is the case. While Golomb coding fails to achieve any significant improvement with relabeling, Elias-δ and Elias-γ coding use considerably less bits than without relabeling. With the FMClustering algorithm, Elias-δ uses 33 % less space on average than the best coding scheme without relabeling, which is the Golomb code. Compression with the Golomb code depends mainly on it's parameter b which in return depends on the sum of all the differences in the posting list. The sum of the differences, however, is left unchanged by relabeling document IDs.

The improvement for the Elias codes comes from the fact, that relabeling with clusters causes most of the gaps between document IDs in a posting list to decrease. For the complete index, Elias-δ is best choice if clusters are used. This changes if only large

posting lists with more than a million elements are considered. With many elements the average gap between document IDs decreases. Because Elias-δ code uses more bits for small numbers and less bits for large numbers in comparison to the Elias-γ code, Elias-γ becomes the better choice when posting lists are large. Therefore, using different encodings for different posting lists could help improve compression even more. Elias-γ coding uses 45 % less space on the GOV2 dataset with clusters and relabeling for posting lists with more than a million elements.

In general the FMClustering algorithm achieves better compressions than the TopDown algorithm. This is in line with the speedups that can be observed for the two algorithms. However the differences are subtle and both achieve considerably better compression than without clustering and relabeling.

7.4.5 Speedups with Comparison-Based Intersection Algorithms

While the Lookup algorithm can intersect sets in $\mathcal{O}(m)$, comparison based intersection algorithms achieve time complexities of $\mathcal{O}(m \log \frac{n}{m})$ on average [BY04]. However this is not a good choice for the cost function directly, as $\log \frac{n}{m}$ approaches 0 as the ratio between the set sizes reaches 1. We therefore use a theoretical lower bound for the number of comparisons

$$\Phi_L(n, m) := \log \binom{n+m}{\min(n, m)}$$

as cost function for intersecting sets. The speedup S_L is calculated as described in Section 7.1 with the new cost function Φ_L. Note that the scoring function for the clustering algorithms still uses the cost function $\Phi(n, m) := \min(n, m)$. The new cost function Φ_L is used solely for evaluation of the speedups.

Figure 7.7 shows the speedup S_L for comparison based algorithms in comparison to the theoretical speedup S_T for the Lookup algorithm on different datasets. With exception of GOV2-TD, which uses the TopDown algorithm, all clusterings are produced by the FMClustering algorithm. The number of clusters is set to $K = 1280$ and the shrink factor to $SF = 0.1$.

The speedup S_L is consistently higher than the speedup S_T on all datasets. On the pagenstecher.de dataset the speedup S_L is even as high as 6, twice the value of the speedup S_T. On the other datasets the differences between the two speedups are subtle. Even if the speedup S_L is only an indication for realistic speedups, comparison based algorithms should also benefit from clustering. Further experiments have to show which speedups can be achieved in practice.

7.4.6 Query Language Model versus Document Collection Language Model

Section 4.3.2 discusses how to determine the probability distribution of the unigram language model used in the scoring function. Two methods are presented. The first uses a query log and the second estimates the language model from the document collection itself.

Figure 7.8 illustrates the speedups on the GOV2 dataset for $K = 2500$ clusters using the TopDown algorithm and a shrink factor $SF = 0.1$. The plots are divided by speedup type. Each plot shows the speedups achieved when using the document collection language model (DC-LM) and the query language model (Q-LM).

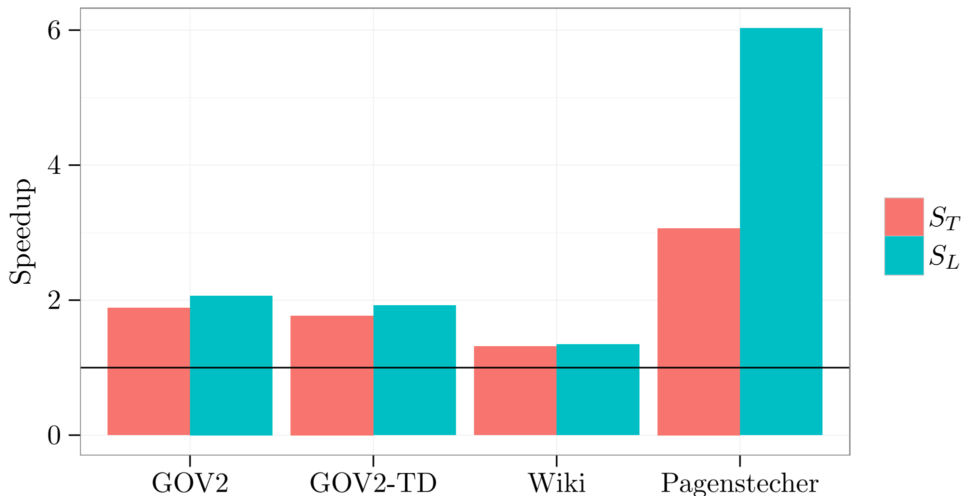

Figure 7.7: Speedups S_T and S_L on different datasets using the adapted cost function for S_L. The clusterings have $K = 1280$ clusters and use the FMClustering algorithm with exception of GOV2-TD.

The clustering is run 10 times for each language model. The AOL query log is divided into 10 parts. For the query language model, 9 parts of the log are used to learn the language model. The remaining part is used to benchmark the speedups. For each run, a different part is used for benchmarking. Since no query log is necessary to determine the document collection language model, the query log is only used for benchmarking. As with the query language model, each run uses a different part of the query log for benchmarking.

The query language model works better for all three speedup types. This is not surprising since the query language model fits the actual queries better, while the document collection language model assumes that queries will use the same language model as the document collection. This is not necessarily the case and so the scoring function leads to a clustering that is suboptimal for the given queries. Nevertheless, in the absence of a query log, using the document collection to generate the language model still yields comparable results. The median for all speedups is less than 6 % worse compared to the query language model.

7.4.7 Clustering Stability

The large amount of data prohibits a complete evaluation of all configurations. That is why the analysis in this chapter is based on representative examples. However, the usefulness of an example depends heavily on how reproducible the results are.

This section gives insights into how reproducible the results are. The speedups on the GOV2 dataset for $K = 2500$ clusters are plotted in Figure 7.8. Each configuration is run 10 times. More details on the parameters are given in Section 7.4.6. The same data, for the query language model, is listed in Table 7.4

The TopDown algorithm generally produces very stable results. The variances range from 1.0×10^{-4} for S_T to 2.0×10^{-4} for S_C when using the query-log language model. Because each run uses a different part of the query log for testing and training, the variances

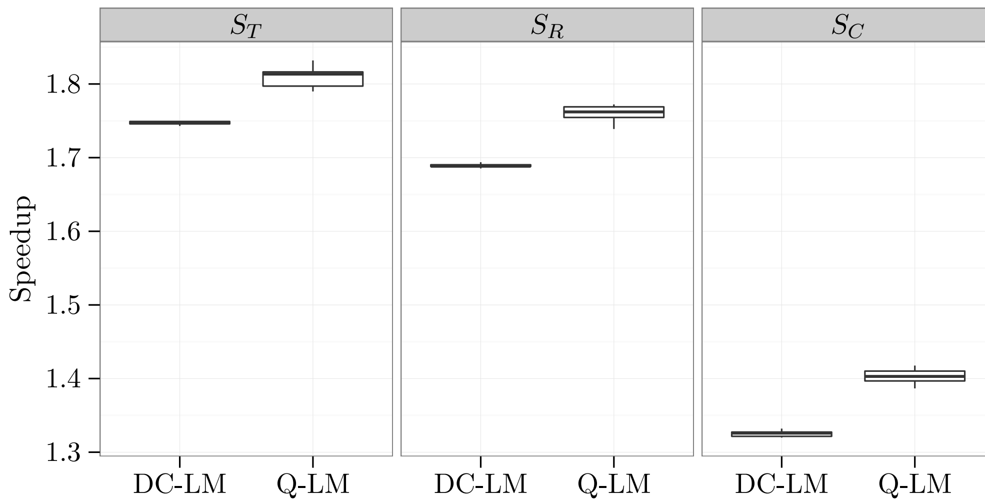

Figure 7.8: Boxplots for the different speedups on GOV2 using the AOL query log language model (Q-LM) and using the document collection language model (DC-LM). $K = 2500$ clusters are used with a shrink factor of $SF = 0.1$

	S_T	S_R	S_C
Minimum	1.790	1.739	1.387
Median	1.813	1.762	1.403
Max	1.832	1.772	1.418
Mean	1.810	1.759	1.403
Variance	2.0×10^{-4}	1.5×10^{-4}	1.0×10^{-4}

Table 7.4: Statistics for the speedups S_T, S_R and S_C for the GOV2 dataset after 10 different clustering runs with the same parameters on different parts of the query log. $K = 2500$ clusters are used with a shrink factor of $SF = 0.1$

are higher than when using the document collection language model. For the document collection language model, where the language models is the same for all runs, the variance decreases to 9.1×10^{-6} for S_T.

This result is important for the validity of the other experiments. Even though there is only one run per configuration, the low variance for the speedups make it a representative example. Big differences between runs are very improbable.

7.4.8 Influence of Term-Cutoff

Figure 7.9 plots the influence of the Term-Cutoff parameter on speedups that are achieved on the GOV2 dataset. The TopDown-Algorithm is used with $K = 2500$ clusters and a shrink factor of $SF = 0.1$. The x-axis uses a logarithmic scale.

The results remain stable for all three speedup measures up until about $TC = 5000$. With less terms the speedups decrease quickly. The 1000 most common terms make up for 55 % of all term occurrences in the GOV2 collection. This increases to about 90 % for the

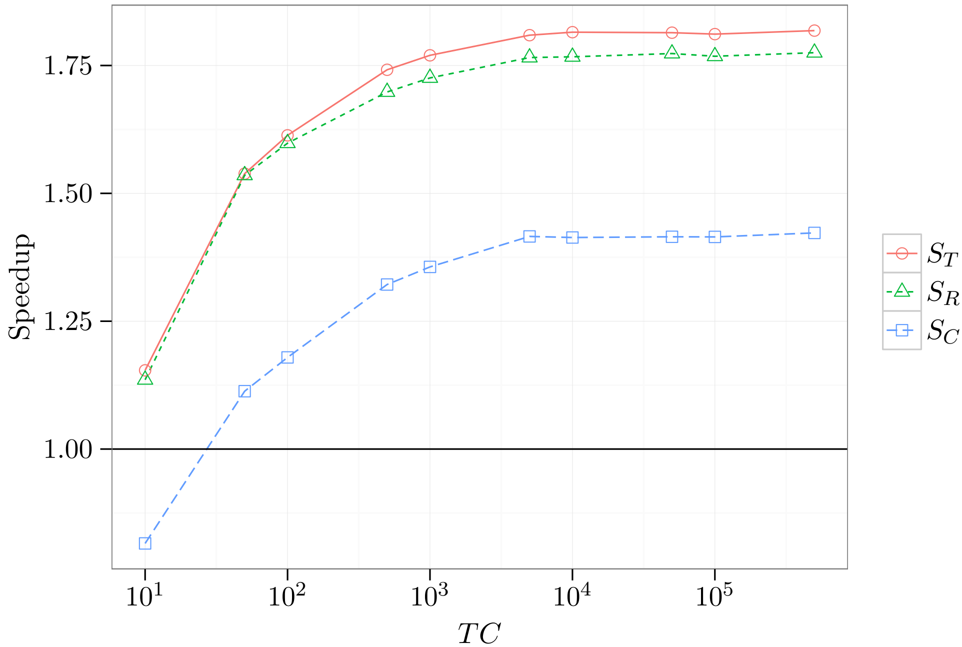

Figure 7.9: Speedups for GOV2 with varying TC. Clustering is produced by the TopDown-Algorithm with $K = 2500$ and $SF = 0.1$.

10 000 most common terms in GOV2. For the clustering this means that with using a Term-Cutoff of $TC = 10\,000$ roughly 90 % of the terms occurring in a document will be used to assign a cluster. The figure shows that this is enough to produce consistent and good results. However, using less terms quickly decreases speedups. The speedups for S_T and S_R still are above 1, even with just 10 terms. This is because the ten most common terms have a big influence on the speedup. Firstly, posting lists are longer for common terms and therefore take longer to intersect in general. With longer intersection times, the influence on the speedup grows. Secondly, common terms are common in queries, increasing the influence even more. However, for the speedup S_C the overhead from the data structure is too high, and 10 terms are not enough to reach speedup greater than 1.

This result has positive implications for the clustering algorithm. As discussed in Section 4.5.2, pre-calculation for the scoring function runs in $\mathcal{O}(K|T|)$ and uses $\mathcal{O}(K|T|)$ space. With a small Term-Cutoff of $TC = 10\,000$ this is nearly 4000 times faster compared to using all terms for GOV2. Regarding the speedups however, there is no noticeable difference.

7.4.9 Influence of Number of Documents

Figure 7.10 shows the speedups as a function of the number of documents $|D|$ used in the clustering. For this experiment $|D|$ random documents are chosen from the GOV2 dataset and clustered using the TopDown algorithm with $K = 2500$ clusters and a shrink factor of $SF = 0.1$. The term-cutoff is set to $TC = 10\,000$.

The speedups show a tendency to increase with the number of documents. While this effect is only small for S_T and S_R, the cluster speedup S_C profits greatly from an increased

number of documents. For $|D| = 5 \times 10^6$ documents, the cluster speedup is as low as $S_C = 1.14$, this increases to $S_C = 1.41$ for $|D| = 25 \times 10^6$.

While more documents seem to be better for the speedups in general, the large increase for S_C can be explained with the overhead for setting up the intersection algorithm for each cluster. With more documents, the posting list in each cluster will be larger. With larger posting lists and longer intersections, the overhead for setting up the clustering algorithm has less influence on the total running time.

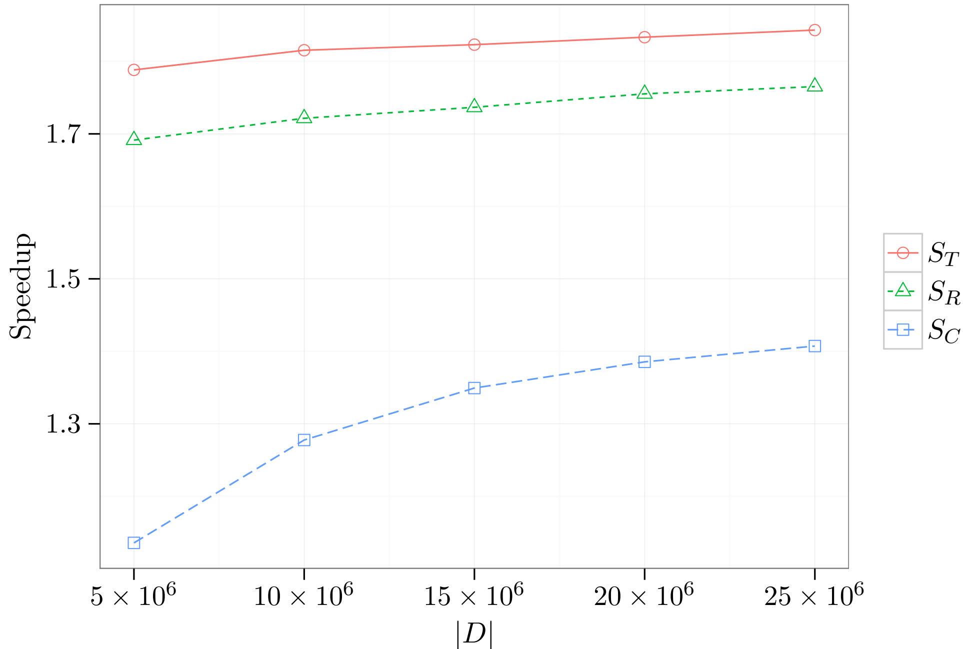

Figure 7.10: Speedups for GOV2 with varying number of documents $|D|$ for $K = 2500$ clusters. The shrink factor is set to $SF = 0.1$.

7.4.10 Speedups for Single Queries

A interesting question is how the speedup depends on the original intersection time. Figure 7.11 shows a scatter plot for the speedups for 5000 randomly sampled two-term queries from the AOL query log. The line in red marks a speedup of 1, that is, no speedup at all. The queries are run on the GOV2 dataset with $K = 2500$ clusters produced by the TopDown algorithm. Each circle represents one query and is plotted with a transparency value of $\alpha = 0.1$.

The speedup varies greatly between queries. Even if the speedup of a single query might be influenced by many factors, like processor utilization or cache lines, there is a clear tendency for higher speedups for longer running set intersections. While some queries show no or even negative speedup if the intersection times are very short, there are virtually no slower queries for longer intersections. This is especially good for the latency of a query. Longer queries, with generally higher latency, tend to achieve a higher speedup.

A scatter plot with the same configuration for the cluster speedup S_C is displayed in Figure 7.12. The increase in speedup for longer intersection times shows in an even more

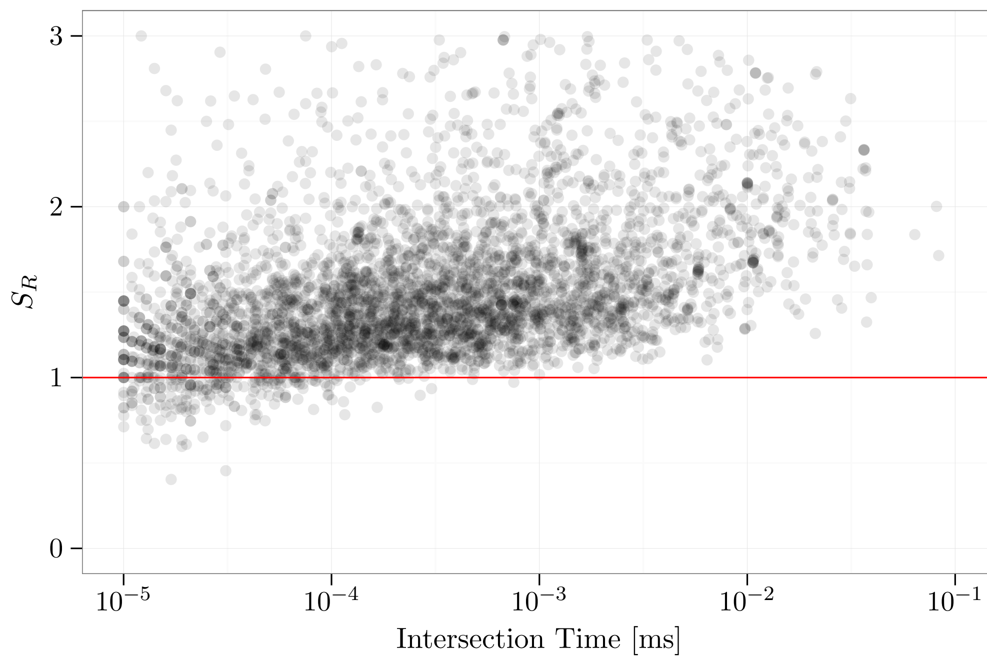

Figure 7.11: The relabeled speedup S_R as a function of the original intersection execution time on the GOV2 dataset with $K = 2500$ clusters. The red horizontal line marks no speedup.

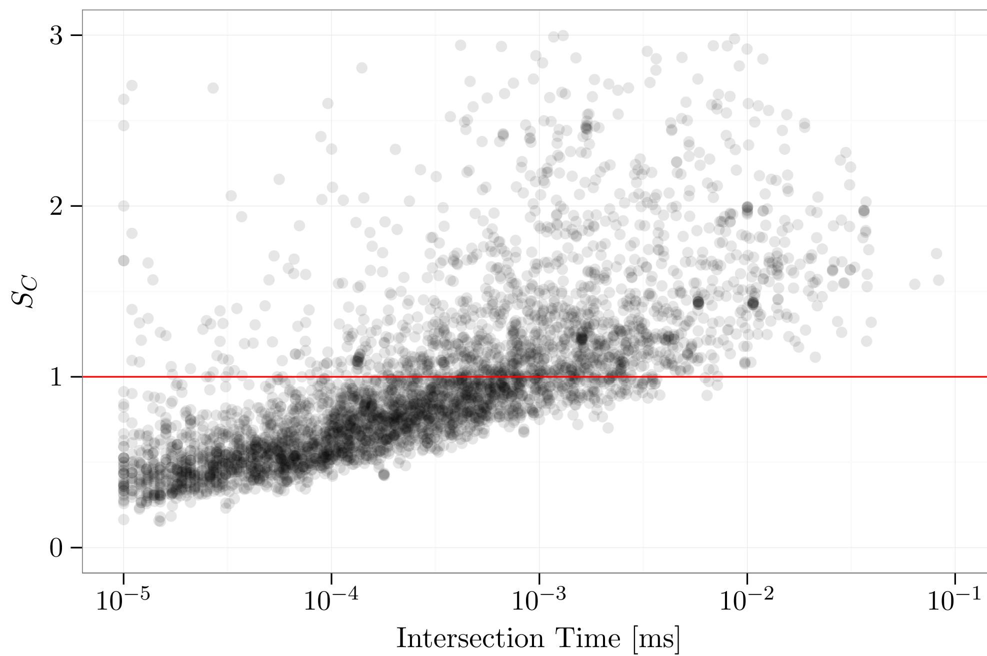

Figure 7.12: The cluster speedup S_R as a function of the original intersection execution time on the GOV2 dataset with $K = 2500$ clusters. The red horizontal line marks no speedup.

obvious way. For very short intersections there are virtually no speedups. The overhead introduced through the Cluster-Lookup algorithm effaces the theoretically better time complexities. However, with longer intersection times, the overhead in general becomes marginal and the speedups increase.

7.4.11 Exact Scoring Function

Section 4.5.3 discusses how the α_2 and α_4 terms from the scoring function can be omitted to increase scoring speed. Figure 7.13 plots the influence of omitting the α terms from the scoring function. The same procedure that is described in Section 7.4.7 is used. The query log is split into 10 parts where 9 parts are used for training and the remaining part is used for testing. This is repeated 10 times, where each part is used once for testing. The clustering is produced by the TopDown algorithm with $K = 2500$ and $SF = 0.1$. The normal scoring function is referred to as *Exact*, the scoring function where α_2 and α_4 are omitted is referred to as *Fast*.

An interesting observation is that the exact scoring function produces slightly worse results for all three evaluation measures, most noticeable for the theoretical speedup S_T and relabeled speedup S_R. The inexact scoring function can be the better choice for both speed and quality of the clustering. However, the TopDown algorithm has less clusters to choose from compared to the FMClustering algorithm, which might decrease the influence of which scoring function is used. Nonetheless we do not have an explanation why the inexact scoring function produces better results than the exact scoring function.

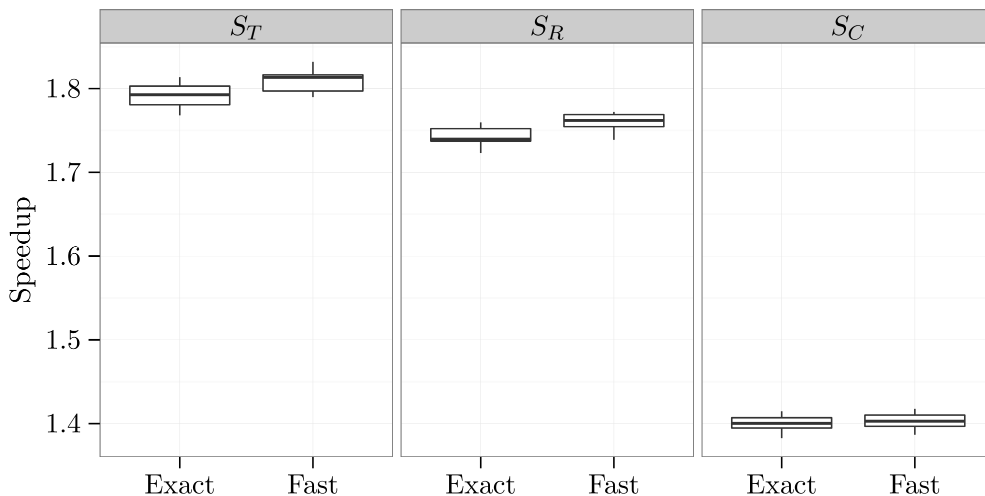

Figure 7.13: Boxplots show the difference between the *Exact* and the *Fast* scoring function for the three different speedups S_T, S_R and S_C. The TopDown algorithm is run 10 times on the GOV2 dataset with $K = 2500$, $SF = 0.1$ and $TC = 10\,000$

7.5 Clustering Runtimes

This section discusses how fast the clustering algorithms are in practice and the influence of various parameters on the runtimes. While a fast implementation of the algorithms is

not a prime concern of this thesis, the experiments show that the algorithms and their implementations run well on large document collections.

The experiments are all conducted on the machine described in Section 7.2.1 and use 16 cores for parallel execution. Since the runtimes vary greatly between individual runs, the experiments are repeated several times in general.

Figure 7.14 and Figure 7.15 show the clustering runtimes as a function of K for the FMClustering and the TopDown algorithm, respectively. The algorithm is executed 10 times on the GOV2 dataset using a Term-Cutoff of $TC = 100\,000$ and shrink factor $SF = 0.1$ for a fixed number of clusters K. The red line connects the arithmetic mean values from the 10 iterations for each K.

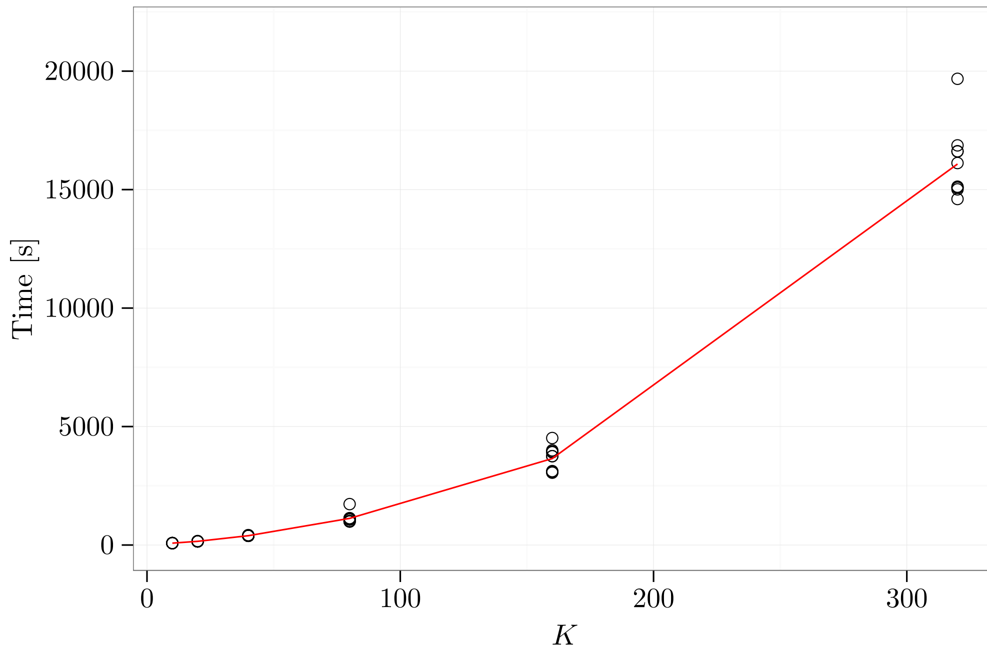

Figure 7.14: Clustering runtimes for increasing number of clusters K on the GOV2 dataset with the FMClustering algorithm. The shrink factor is set to $SF = 0.1$ and the term cutoff to $TC = 100\,000$. The clustering algorithm is run 10 times for each K

For the FMClustering algorithm in Figure 7.14 there is a super-linear increase of the average runtime for increasing K. The time complexity analysis in Section 4.4.1 gives a lower bound that is linear in the number of clusters. While the depth of the recursion is deterministic, the number of iterations in each recursion level is not. With an increasing number of clusters the algorithm needs more time to converge to a stable solution. The result is a super-linear increase of the average runtime. A large number of clusters can therefore become infeasible for the FMClustering algorithm.

In contrast, the TopDown algorithm does not show this super-linear increase. The runtimes for the TopDown algorithm are plotted in Figure 7.15. The plot shows a sub-linear increase of the average runtimes for increasing number of clusters K. Since the quality of the clustering, with regard to the achievable speedup, increases with the number of clusters, this is an important observation.

For $K = 2560$ clusters, the TopDown algorithm takes on average 1477 s, or just short of half an hour, to cluster the GOV2 dataset. This corresponds to a average time of 59 µs per document. $K = 2560$ clusters is a value that works well on the GOV2 dataset. The processing and indexing of a document in our framework is in the order of several milliseconds per document. Even if the processing is not highly optimized, clustering with the TopDown algorithm has a negligible influence on the total processing time. A technique like ID relabeling, as discussed in Section 5.2, can therefore be used without a large impact on preprocessing time.

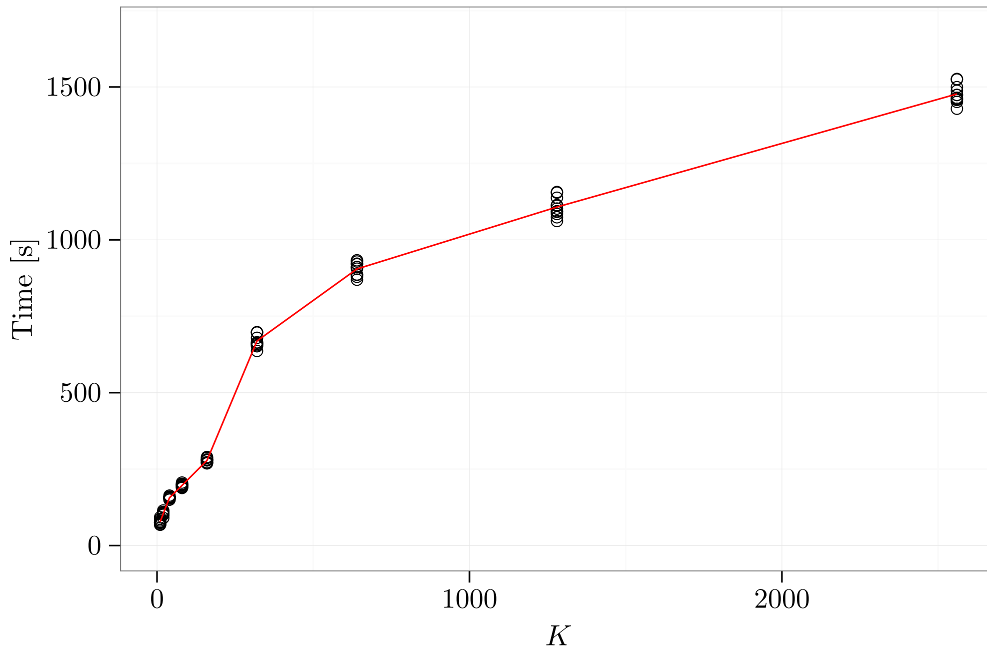

Figure 7.15: Clustering runtimes for increasing number of clusters K on the GOV2 dataset with the TopDown algorithm. The shrink factor is set to $SF = 0.1$ and the term cutoff to $TC = 100\,000$. The clustering algorithm is run 10 times for each K

7.5.1 FMClustering versus TopDown

There is a vast difference between the runtimes of the TopDown algorithm and the FMClustering algorithm. A direct comparison is difficult, since the runtimes for both algorithms show different behavior for increasing number of clusters. However, the rule of thumb is that the TopDown algorithm performs more than 10 times faster than the FMClustering algorithm. With an increasing number of K the discrepancy between the runtimes increases even more.

This leads to a simple tradeoff between query and preprocessing time. The FMClustering algorithm leads to somewhat better results. However the long runtimes can be prohibitive. If a quick preprocessing of the document collection is more important, the TopDown algorithm achieves very good clustering times with reasonable quality for the clustering.

7.5.2 Influence of Term-Cutoff

The influence of the Term-Cutoff parameter on the clustering runtimes is plotted in Figure 7.16. The plot shows the runtimes as a function of the Term-Cutoff parameter TC on the GOV2 dataset for $K = 100$ clusters.

The Term-Cutoff parameter has a big influence on the runtimes of both clustering algorithms. While the runtimes do not increase linearly with the Term-Cutoff parameter, runtimes decrease significantly with a lower Term-Cutoff. Section 7.4.8 shows that the choice of the Term-Cutoff parameter has no influence on the quality of the clusterings above a certain threshold. Implementations that are interested in fast clusterings with the best possible quality should therefore choose a value for the Term-Cutoff parameter that comes close to the threshold value. Choosing a low Term-Cutoff value also decreases space requirements of the algorithm.

7.5.3 Influence of Document Collection Size

Figure 7.17 shows the influence of the number of documents to cluster on the runtimes of the two algorithms. The algorithms use $K = 250$ clusters with a shrink factor of $SF = 0.1$ and Term-Cutoff of $TC = 10\,000$ on randomly sampled documents from the GOV2 document collection.

From $|D| = 3 \times 10^6$ upwards there is an obvious linear tendency for the runtimes for both algorithms. However, both algorithms show a local maximum at $|D| = 2 \times 10^6$. This is an artifact introduced by the constant threshold described in Section 4.4.2. If the number of documents is less than $K \cdot 100$, the mode changes from multi-threaded iteration-grained execution to single-threaded document-grained iteration. While the document-grained iteration can be implemented with multi-threading support, the implementation in this thesis uses single-threaded execution for simplicity. Because of the shrink factor $SF = 0.1$ and the number of clusters $K = 250$, the number of recursions that use iteration-grained pre-calculation is higher when the number of documents is smaller than $|D| < 2.5 \times 10^6$. With a higher number of single-threaded document-grained recursion levels, the algorithms also run slower, resulting in the visible local maximum at $|D| = 2 \times 10^6$. Document-grained pre-calculation is generally slower and single-threaded execution further intensifies this. With a higher resolution on the x-axis, there would be a sharp drop of runtimes at $|D| = 2.5 \times 10^6$. Nonetheless, the linear tendency holds when the number of recursion levels that use document-grained pre-calculation stays constant.

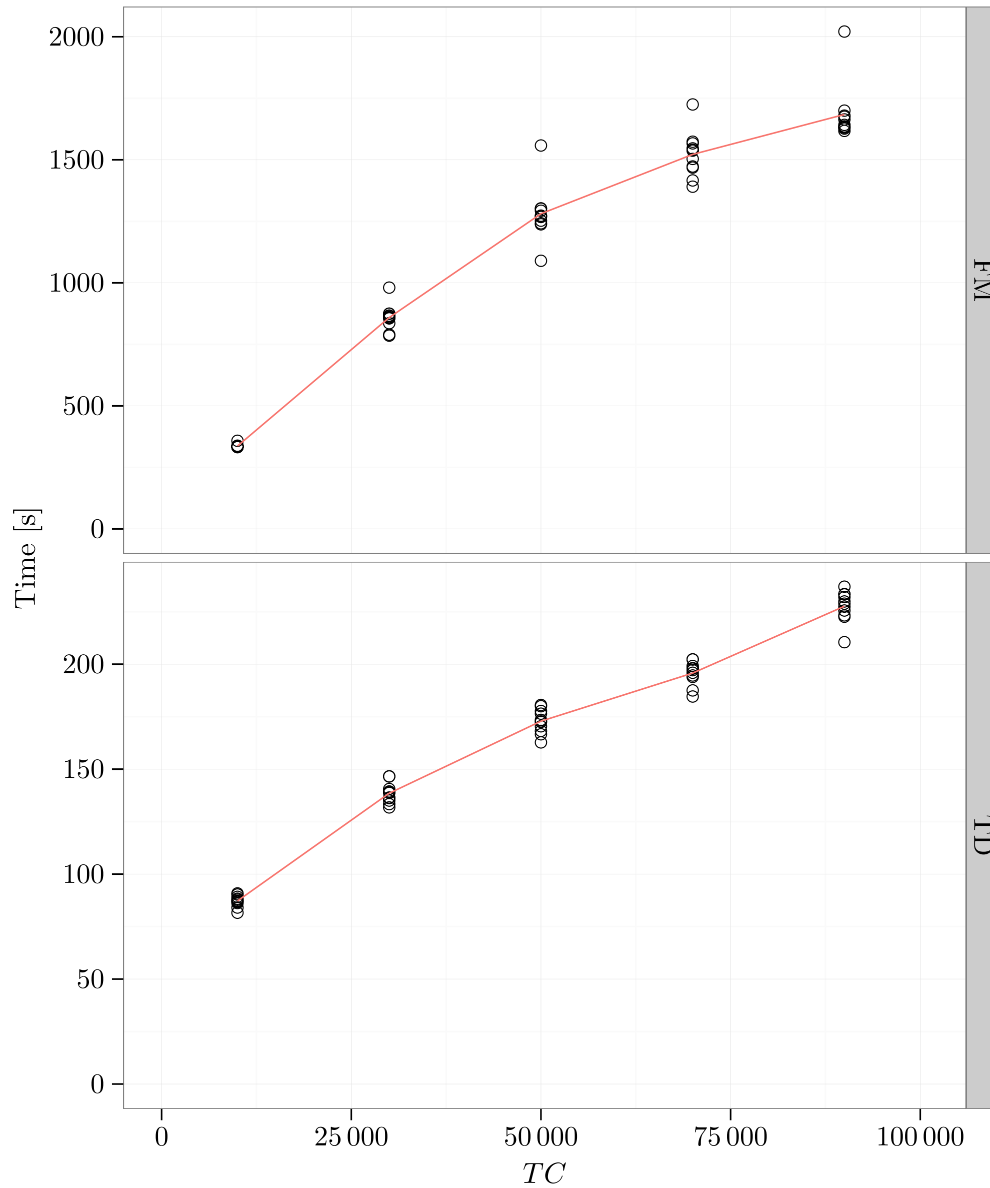

Figure 7.16: Clustering runtimes as a function of the Term-Cutoff parameter TC on the GOV2 dataset with $K = 100$ and $SF = 0.1$

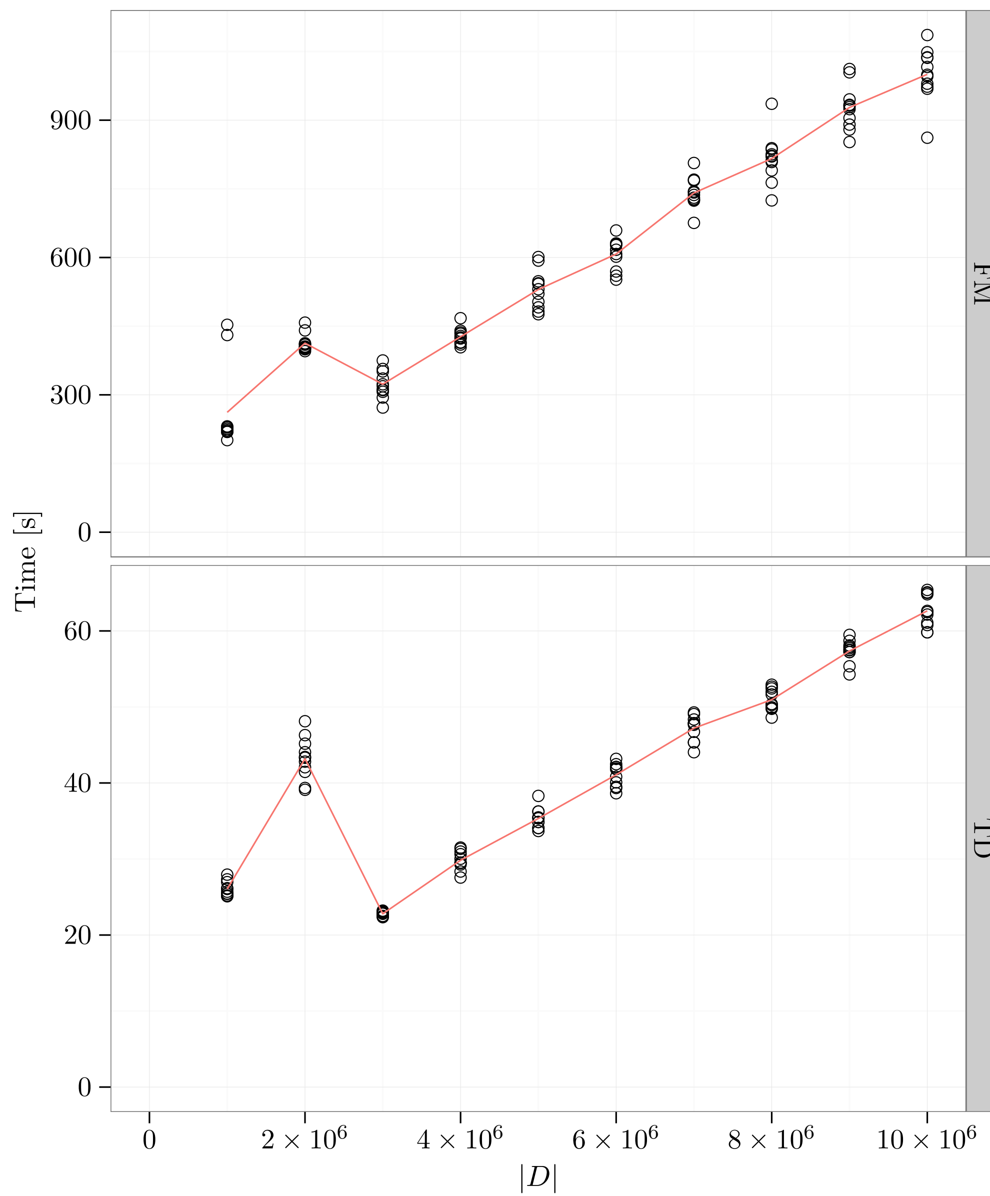

Figure 7.17: Clustering runtimes as a function of the number of documents to cluster $|D|$ on the GOV2 dataset with $K = 250$, $SF = 0.1$ and $TC = 10\,000$

8. Conclusion

This chapter summarizes this thesis and gives an outlook on future work.

8.1 Summary

The work on this thesis is based on an efficient and effective scoring function for assigning documents to clusters. The scoring function explicitly minimizes the expected time complexity of conjunctive queries. To the best of our knowledge, this thesis is the first to optimize the amount of work for intersecting posting lists by document clustering while maintaining accuracy.

The thesis shows that the scoring function can be calculated very efficiently in $\mathcal{O}(|d|)$ time for each cluster and document with simple pre-calculated data structures. The thesis further discusses two fast and scalable clustering algorithms suitable for document clustering. The algorithms use the scoring function to greedily assign documents to clusters. Clusters are refined through iteratively reassigning documents to the cluster with the best score. On the GOV2 dataset clustering is as fast a 60 µs per document on current commodity hardware.

Two techniques that use a clustering of documents to speedup conjunctive queries are presented and analyzed. The first uses a new data structure to improve the actual time complexity of posting list intersections. While the first comes along with considerable overhead, the second technique uses relabeling of document IDs which does not introduce any overhead for intersections. The technique does not yield a better time complexity in general, but experiments show that the technique achieves a considerable speedup in practice.

A systematic evaluation of the clustering algorithms on real world data rounds the work of this thesis off. Theoretical and actual speedups are examined on multiple real world datasets. The experiments give insights into the various tuning parameters of the clustering algorithms. On the widely used GOV2 dataset, posting list intersections can be close to 2 times faster with document clustering than without. On a document collection that concentrates on a specialized topic a real speedup of over factor 4 can be measured. Evaluation of the clustering algorithm runtimes show that clustering even large document collections is feasible with the approaches presented in this thesis.

8.2 Future Work And Outlook

Because this thesis is the first to use document clustering to minimize the intersection times of posting lists, a new range of interesting research questions emerge.

Implementation

The implementation of the Cluster-Lookup algorithm is kept simple. Using fine-tuned and benchmarked code could improve the speedup and reduce overhead. For example, while the Lookup algorithm provides the best time complexity, the Zipper algorithm can perform faster in practice [Tra10]. For clusters with only a couple of documents for a term, using a simpler algorithm than Lookup could increase performance.

The implementations for the FMClustering and TopDown algorithms leave room for further improvements. Both implementations contain sequential code that can be parallelized.

Compression

In-memory full text search engines utilize compression to minimize the amount of memory necessary for posting lists [Tra10]. While this thesis shows that document clustering can improve posting list compression, the question remains how the real speedups change when compression is involved.

Queries

The analysis in this thesis is limited to two-term conjunctive queries. However in practice queries often contain multiple terms. Moreover queries can also be disjunctive, where a document matches a query if it contains any term from the query. An interesting open research topic is how the speedup changes if more than 2 terms occur in a conjunctive query. Research is needed to determine whether disjunctive queries can benefit from document clustering. Ranking results, top-k and phrase queries are important techniques in the area of information retrieval. A question is how document clusters can also benefit top-k and phrase queries or how ranking the documents influences the speedups.

Other Set Intersection Algorithms

There exist a variety of set intersection algorithms. The analysis in this thesis is limited to the Lookup [ST07] algorithm because it has a time complexity that is linear in the size of the smaller set. Ding and König [DK11] present an algorithm based on hashes that can perform faster than Lookup under some circumstances. The theoretical speedup that can be achieved when using document clustering indicates that the approach from this thesis can also be applied to other state-of-the-art posting list intersection algorithms.

Clustering Algorithms

The scoring function presented in this thesis is not limited to the algorithms discussed. While the algorithms in this thesis are chosen to fit the problem well, other algorithms might produce better results or execute faster.

Distributed Search

Large document collections are usually handled by using distributed search, and partitioning either terms or documents among shards. Section 5.4 handles the applicability of document clustering to distributed search. However no experimental analysis is performed in this thesis. An open question is which speedups can be achieved when using distributed search.

Bibliography

[BDH03] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *Micro, Ieee*, 23(2):22–28, 2003.

[Ber06] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[BY] Ricardo Baeza-Yates. Web log mining in search engines.

[BY04] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, pages 400–408. Springer, 2004.

[BYRN+99] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[CG96] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[CKL+07] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[Con13] Library Of Congress. Update on the twitter archive at the library of congress. White Paper, 2013.

[CPBY09] B Barla Cambazoglu, Vassilis Plachouras, and Ricardo 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. ACM, 2009.

[CSC04] Charlie Clarke, Ian Soboroff, and Nick Craswell. Gov2 test collection, 2004. URL: `http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DK11] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proceedings of the VLDB Endowment*, 4(4):255–266, 2011.

[DMdSF+05] Edleno S De Moura, Célia F dos Santos, Daniel R Fernandes, Altigran S Silva, Pavel Calado, and Mario A Nascimento. Improving web search efficiency via a locality based static pruning method. In *Proceedings of the 14th international conference on World Wide Web*, pages 235–244. ACM, 2005.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[Gol66] Solomon. W. Golomb. Run-length encodings. *IEEE Trans Info Theory*, 12(3):399–401, 1966.

[JHA+99] S Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, et al. Haskell 98: A non-strict, purely functional language, 1999.

[JO95] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(2):142–153, 1995.

[KFN10] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 441–450. ACM, 2010.

[Knu92] Donald E Knuth. Two notes on notation. *The American Mathematical Monthly*, 99(5):403–422, 1992.

[Kul13] Anagha Kulkarni. Efficient and effective large-scale search. 2013.

[LOPS07] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Proceedings of the 2nd international conference on Scalable information systems*, page 43. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967.

[MMR00] Andy MacFarlane, Julie A McCann, and Stephen E Robertson. Parallel search using partitioned inverted files. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 209–220. IEEE, 2000.

[PCT06] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st international conference on Scalable information systems*, page 1. Citeseer, 2006.

[Per94] Michael Persin. Document filtering for fast ranking. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 339–348. Springer-Verlag New York, Inc., 1994.

[PSL06] Diego Puppin, Fabrizio Silvestri, and Domenico Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st*

international conference on Scalable information systems, page 34. ACM, 2006.

[SC99] Fei Song and W Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321. ACM, 1999.

[SKK+00] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

[ST07] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, volume 7, 2007.

[Tra10] F Transier. *Algorithms and Data Structures for In Memory Text Search Engines.* PhD thesis, PhD thesis, University of Karlsruhe, 2010.

[VR79] C.J. Van Rijsbergen. *Information retrieval.* Butterworths, 1979.

[XC99] Jinxi Xu and W Bruce Croft. Cluster-based language models for distributed retrieval. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261. ACM, 1999.