Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau

Efficient Error-Tolerant Search on Large Text Collections

Marjan Celikik

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Dekan

Prof. Dr. Bernd Becker

Referenten

Prof. Dr. Hannah Bast

Priv.-Doz. Dr. Ralf Schenkel

Datum der Promotion: 09.09.2013

Contents

Abstract

In this dissertation, we consider the problem of fuzzy full-text search and query suggestion in large text collections, that is, full-text search that is robust against errors on the side of the query as well as errors on the side of the documents. We consider two variants of the problem. The first variant is keyword-based search tolerant to errors. The second variant is autocompletion or prefix search tolerant to errors. In this variant of the problem, each keyword can be specified partially and the results appear instantly as the user types the query letter by letter.

One of the main challenges in building an information retrieval system for fuzzy search is efficiency. Providing interactive query times (below 100 ms) for either fuzzy search variant is surprisingly challenging due to the one order of magnitude larger volume of data to be handled by the system. While efficient index data structures exist that allow fast search for the exact variants of the problem, there has been limited work on indexes that tackle fuzzy search.

Commercial search engines such as Yahoo!, Google and Bing provide error-tolerance to certain extent thanks to the large amount of available query log data. In our version of the problem, we assume a complete absence of query logs or any other precomputed information. This assumption is often realistic for information retrieval systems for vertical or domain-specific search that typically have a much smaller user base.

In the first part of this dissertation, we propose efficient data structures and algorithms that are the core of our fuzzy search. In the second part, we address important algorithm-engineering aspects of an error-tolerant search system. All of our algorithms and data structures have been implemented and integrated into the CompleteSearch engine.

Zusammenfassung

In Richtung Effiziente und Vollständig Fehlertolerante Suche

In einem traditionellen Informationssystem wird vom Benutzer erwartet, eine vollständige Suchanfrage im Voraus im Sinne von unzweideutigen und exakten Schlüsselworten zu spezifizieren. In vielen Suchanwendungen hat der Benutzer über die zugrundeliegenden Daten jedoch nur unvollständiges Wissen. Ein moderner Trend um dieses Problem zu verringern, ist Suche die tolerant bezüglich Tippfehlern ist. Im Folgenden werden wir uns dieses Paradigma der Informationssuche näher anschauen und den aktuellen Stand der Wissenschaft sowie seine derzeitigen Herausforderungen diskutieren.

Fehlertolerante Schlüsselwortbasierte Suche

Toleranz der Anfrageseite: Der Leser ist vielleicht schon damit vertraut, dass es erstrebenswert (und oft notwendig) ist, dass eine Suchmaschine robust gegen Fehler in der Anfrage ist. Zum Beispiel, wenn ein Benutzer `probablistic database` tippt, soll die Suchmaschine eine nach Rang geordnete Liste von Dokumenten zurückliefern, welche die Schlüsselworte `probabilistic database` enthalten. Web-Suchen wie Google bieten diese Eigenschaft durch Rückfrage "Did you mean: probabilistic database" oder durch Vorschlagen von alternativen Anfragen in einer drop-down Box.
Toleranz der Dokumentseite: Nehmen wir das Gegenteil an, dass unsere Schlüsselworte korrekt sind; ein relevantes Dokument in unserer Kollektion enthält jedoch einen Rechtschreibfehler. Unsere Informationssuche wäre immer noch unerfolgreich. Dies passiert überraschend oft, entweder durch vom Autor des Dokumentes verursachte Tippfehler oder aufgrund von OCR Fehlern. Web-Suchmaschinen wie Google bieten diese Eigenschaft nicht. Der Hauptgrund ist, dass es in Web-Suchen für gewöhnlich mehr als genug Treffer gibt und es das primäre Ziel ist, den besten dieser Treffer zu finden. In gebietsspezifischen Suchen ist jedoch *recall* (Anteil der relevanten Dokumente die wiedergefunden werden) genauso wichtig wie *precision* (Anteil der gefundenen Dokumente die relevant sind) und manchmal sogar wichtiger.

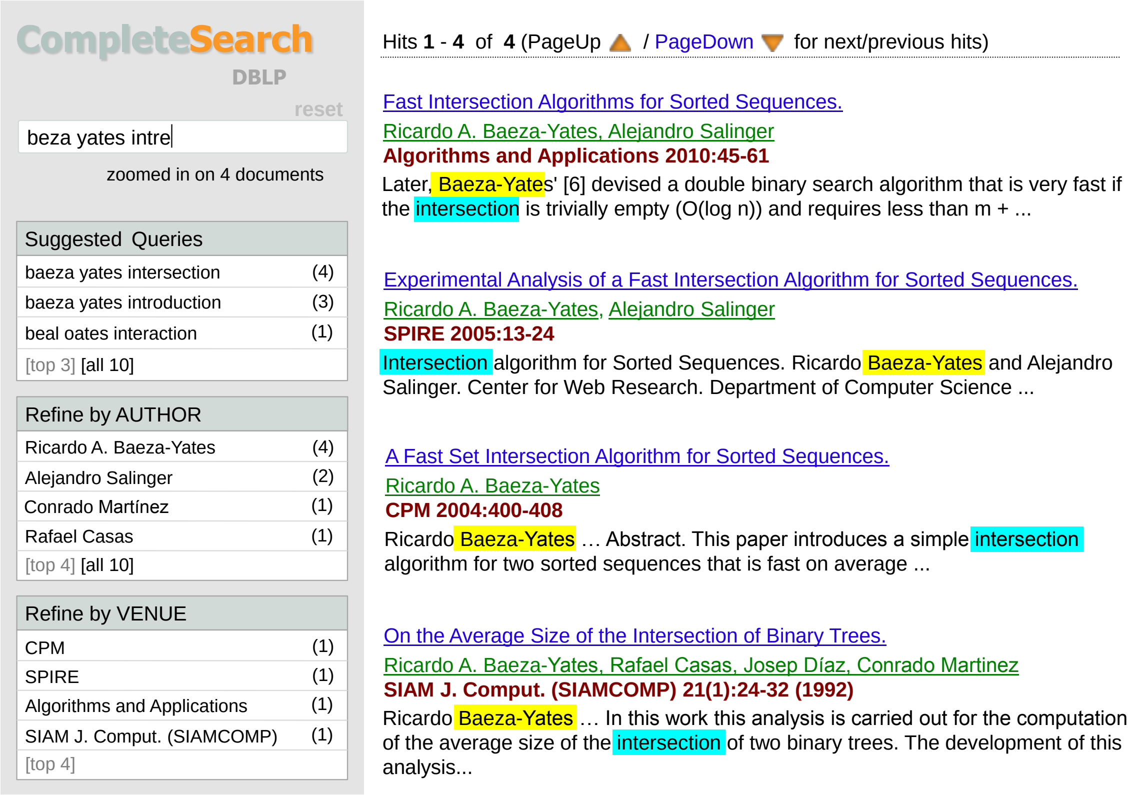
Fehlertolerante Autovervollständigungssuche

Heute ist Autovervollständigung eine ubiquitäre Eigenschaft, die in vielen Suchanwendungen als nützlich empfunden wird. Die fehlertolerante Variante der Autovervollständigungssuche ist ein weiterer aufkommender Trend in der modernen Informationssuche. In diesem Suchparadigma, in dem jeder Buchstabe getippt wird, verlangen wir augenblickliches Darstellen der Treffer und der Vervollständigungen welche das Präfix näherungsweise enthalten. Man betrachte zum Beispiel den Namen `Tchaikovsky`. Es ist wahrscheinlich, dass ein Benutzer bereits ein fehlerhaftes Präfix von diesem Schlüsselwort tippen wird, zum Beispiel `Tschai` or `Tczai`. Andererseits, wenn kurze Präfixe wie `T` getippt werden, ist es wahrscheinlich, dass die Zahl der zurückgegebenen Vervollständigungen zu groß ist um nützlich zu sein.

Vorschläge von Suchanfragen

Es ist wünschenswert, dass dem Benutzer eine Menge von Suchanfragevorschlägen zusammen mit den Suchergebnissen gezeigt werden, insbesondere wenn die Anzahl der Treffer klein ist . Man betrachte die Anfrage `beza yates intre`. Neben anderen sollte diese Suchanfrage in den Dokumenten resultieren, welche die Schlüsselwörter `baeza yates intersection` enthalten. Dieses Beispiel demonstriert das Problem der Berechnung der passendsten Suchanfragevorschläge. Abbildung 0.1 zeigt ein Bildschirmphoto von unserer fehlertoleranten Autovervollständigungssuche beim Durchsuchen der DBLP Kollektion[1] von Informatikartikeln.

[1] `dblp.uni-trier.de`, `www.dblp.org/search`


CompleteSearch
DBLP

reset

beza yates intre

zoomed in on 4 documents

Suggested Queries	
baeza yates intersection	(4)
baeza yates introduction	(3)
beal oates interaction	(1)
[top 3] [all 10]	

Refine by AUTHOR	
Ricardo A. Baeza-Yates	(4)
Alejandro Salinger	(2)
Conrado Martínez	(1)
Rafael Casas	(1)
[top 4] [all 10]	

Refine by VENUE	
CPM	(1)
SPIRE	(1)
Algorithms and Applications	(1)
SIAM J. Comput. (SIAMCOMP)	(1)
[top 4]	

Hits **1** - **4** of **4** (PageUp ▲ / PageDown ▼ for next/previous hits)

Fast Intersection Algorithms for Sorted Sequences.
Ricardo A. Baeza-Yates, Alejandro Salinger
Algorithms and Applications 2010:45-61
Later, Baeza-Yates' [6] devised a double binary search algorithm that is very fast if the intersection is trivially empty (O(log n)) and requires less than m + ...

Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences.
Ricardo A. Baeza-Yates, Alejandro Salinger
SPIRE 2005:13-24
Intersection algorithm for Sorted Sequences. Ricardo Baeza-Yates and Alejandro Salinger. Center for Web Research. Department of Computer Science ...

A Fast Set Intersection Algorithm for Sorted Sequences.
Ricardo A. Baeza-Yates
CPM 2004:400-408
Ricardo Baeza-Yates ... Abstract. This paper introduces a simple intersection algorithm for two sorted sequences that is fast on average ...

On the Average Size of the Intersection of Binary Trees.
Ricardo A. Baeza-Yates, Rafael Casas, Josep Díaz, Conrado Martinez
SIAM J. Comput. (SIAMCOMP) 21(1):24-32 (1992)
Ricardo Baeza-Yates ... In this work this analysis is carried out for the computation of the average size of the intersection of two binary trees. The development of this analysis...

Abbildung 0.1.: Eine derzeitige Version von CompleteSearch, welche mit einem unserer neuen Algorithmen fuzzy-Suche auf DBLP betreibt. Die Suchanfrage, die getippt wird, ist *beza yates intre* (die beabsichtigte Suchanfrage lautet *beza yates intersection*, was auf eine Liste von Schnittalgorithmen des Autors Ricardo Baeza-Yates verweist). Es soll angemerkt werden, dass Google derzeit keinen sinnvollen Vorschlag für diese Suchanfrage liefert.

Herausforderungen

Es gibt bereits effiziente Indexdatenstrukturen, welche schnelle und exakte Autovervollständigungssuche erlauben. Für die prominentesten Vorschläge sollte sich der Leser auf Bast and Weber [2006] und Ji *et al.* [2009] beziehen. Es gibt jedoch überraschend wenige Arbeiten über Indexe die effiziente und fuzzy-Schlüsselwortbasierte- oder fuzzy-Autovervollständigungssuche bieten.[2] Google's Web-Suche bietet in ihrem derzeitigem Stand fuzzy-Suche in beiden Varianten, so lange die Autovervollständigung nur auf dem letzten Schlüsselwort stattfindet und so lange die Fehler auf der Seite der Suchanfrage liegen. In Google's Ansatz gibt es jedoch einen fundamentalen Unterschied. Google hat nämlich Milliarden von Suchanfragen als Rohstoff. Sie können ahnen, was der Benutzer gemeint oder beabsichtigt hat, basierend auf Hunderten von ähnlichen Suchanfragen. Als Konsequenz davon bekommt der Benutzer keine augenblickliche Suche sowie keine Suchvorschläge, wenn die Suchanfrage nicht in der vorkompilierten Suchanfragenaufzeichnung vorkommt (z.B. eine Experten-Suchanfrage mit relativ kleiner Treffermenge).

Abwesenheit von Suchanfragenaufzeichnungen

Sich auf populäre Suchanfragen von einer vorkompilierten Liste zu fokussieren ergibt für eine Web-Suche viel Sinn, aber es ist unbefriedigend für eine gebietsspezifische Suchanwendung (z.B. Literatursuche, E-Mailsuche, Benutzeroberflächensuche). Dies ist der Fall, da große Suchanfragenaufzeichnungen selten verfügbar sind und "Experten-Suchanfragen" mit kleinen Treffermengen eher die Regel als die Ausnahme darstellen. Deshalb nimmt unser Ansatz die völlige Abwesenheit von Suchanfrageaufzeichnungen an.

[2] Man nehme zur Kenntniss, dass approximative Zeichenkettenabgleichung, bei der es darum geht alle approximativen Vorkommnisse eines kurzen Musters P in einem langen Text T zu finden, ein besser studiertes, aber schwierigeres Problem ist.

Große Datenmengen

Wenn wir es mit großen Kollektionen (mehr als 100 GB) zu tun haben, wird bei unserer Problemstellung die Effizienz eine große Herausforderung. Interaktive Suchanfragezeiten (unter 100 ms) für die fuzzy-Suche zu bieten wird ein überraschend großes Problem. Der Hauptgrund ist, dass sich die Gesamtzahl der Vorkommen in einer Kollektion, die ähnlich zu einem gegebenen Schlüsselwort oder Präfix ist, schneller als linear mit der Größe der Kollektion erhöht. In der Praxis resultiert dies in einem ungefähr zehnmal größeren Volumen, das vom System ausgewertet werden muss, verglichen mit der exakten Suche. Deshalb wird für Algorithmen die auf vergleichbar kleinen Kollektionen effizient sind, Skalierbarkeit zum Problem. Die Anforderung von interaktiven Suchanfragezeiten macht dieses Problem noch schwieriger.

Entwicklungsherausforderungen

Unsere Arbeit beinhaltet größere Herausforderungen, die technischer und entwicklungsmäßiger Natur sind, was eine sorgsame Algorithmenentwicklung erfordert. Unser erstes Ziel ist die Entwicklung von Algorithmen, die nicht nur in der Theorie effizient sind, sondern auch in der Praxis. Unser zweites Ziel ist die Implementierung eines vollständigen fuzzy-Suchsystems. Solche Systeme beinhalten Entwicklungsherausforderungen, die sich von denen der "gewöhnlichen" Suchsysteme unterscheiden. Dies schließt die Generierung von Textauszügen mit unexakten Treffern, sowie die effiziente Konstruktion von einem Index, der fuzzy-Suche unterstützt, ein.

Außerhalb Textlicher Ähnlichkeit

Es gibt Instanzen des fuzzy-Suchproblems, die von den konventionellen Ähnlichkeitsdistanzansätzen nicht abgedeckt werden können. Eine solche Instanz ist das Wortgrenzenproblem. Dies tritt auf, wenn bestimmte Phrasen manchmal zusammen und manchmal als mehrere Worte geschrieben werden (z.B. dishwasher und dish washer). Speziell relevant ist dies für Sprachen mit zusammengesetzten Nomen, so wie Deutsch. Eine weitere Instanz sind Worte die nur phonetisch ähnlich sind (z.B. lynx und lincks).

Beiträge und Überblick

Der Beitrag dieser Dissertation besteht aus zwei Teilen. Der erste Teil befasst sich mit effizienten Datenstrukturen und Algorithmen die der Kern unserer fuzzy-Suche sind. Der zweite Teil behandelt wichtige technische Aspekte von einer fehlertoleranten Suchmaschine. Alle unsere Algorithmen und Datenstrukturen wurden in die CompleteSearch Suchmaschine Bast and Weber [2006], welche Ergebnisse und Vorschläge liefert während man tippt, implementiert.

Nach dem Formalisieren von jedem Problem präsentieren wir effiziente Wortabgleichungsalgorithmen, welche Fehler erlauben (auch bekannt als approximative Lexikonsuche), gefolgt von effizienten Präfixabgleichungsalgorithmen die Fehler erlauben (auch bekannt als fehlertolerante Autovervollständigung). Diese zwei Probleme sind Voraussetzungen für das zentrale Problem welches in dieser Dissertation betrachtet wird, nämlich Indexe für effiziente fuzzy-Schlüsselwortbasierte- und Autovervollständigungssuche. Dann betrachten wir das Problem von Suchanfragevorschlägen in Abwesenheit von Suchanfrageaufzeichnungen. Die zwei abschließend betrachteten Probleme sind effiziente Indexkonstruktion, gefolgt von effizienter Ergebnisauszuggenerierung für fuzzy-Suche.

Im Folgenden bieten wir eine detailliertere Kapitel-für-Kapitel Aufteilung für alle der oben genannten (Sub)-Probleme, sowie eine kurze Zusammenfassung von ihren Beiträgen.

In Kapitel 2 formalisieren wir die algorithmischen Probleme die in dieser Thesis betrachtet werden. Zusätzlich präsentieren wir für unser zentrales Problem eine ad-hoc Lösung, welche einen standardmäßigen invertierten Index benutzt.

In Kapitel 3 präsentieren wir für das fuzzy-Wortabgbleichungsproblem zwei neue, praktische Algorithmen mit verschiedenen Vorteilen. Abhängig vom Schwellenwert übertreffen unsere Algorithmen die ehemals besten Algorithmen um bis zu Faktor zehn.

In Kapitel 4 zeigen wir wie beide der oben genannten Algorithmen zu effizienten fuzzy-Präfixabgleichungsalgorithmen erweitert werden können. Auch hier erreichen wir eine Verbesserung um bis zu Faktor zehn gegenüber den ehemals besten Algorithmen.

In Kapitel 5 stellen wir zwei neue Datenstrukturen, fuzzy-Wortindex und fuzzy-Präfixindex genannt, sowie einen neuen Suchanfrageverarbeitungsalgorithmus vor. Unsere Algorithmen bieten fuzzy-Volltextsuche, die (1) robust gegen Fehler auf der Anfrageseite, sowie (2) robust gegen Fehler auf der Dokumenteseite ist, die (3) Schlüsselwortbasierte- und Autovervollständigungssuche unterstützt und (4) die auch für große Textkollektionen (bis zu 100 GB) interaktive Suchanfragezeiten auf einer Standardmaschine hat.

In Kapitel 6 fokussieren wir uns auf Algorithmen zur Berechnung einer nach Rang geordneten Liste von Suchanfragevorschlägen und Phrasenvervollständigungen. Unsere Algorithmen benötigen keine Suchanfrageaufzeichnungen und brauchen nur einen kleinen Bruchteil der Suchanfrageverarbeitungszeit.

In Kapitel 7 erörtern wir die relevanten Erweiterungen und Details unserer fuzzy-Suche die in den vorherigen Kapiteln nicht diskutiert wurden.

In Kapitel 8 betrachten wir das Problem der effizienten Indexkonstruktion für fuzzy- sowie exakte Suche. Wir zeigen, dass der Index für unsere fuzzy-Suche so schnell wie ein gewöhnlicher invertierter Index, und bis zu doppelt so schnell für eine exakte Suche, konstruiert werden kann.

In Kapitel 9 sprechen wir das Problem der Generierung von Textauszügen für komplexe Suchanfragen und nicht-buchstäbliche Abgleichungen, wie die von der fuzzy-Suche produzierten, an. Neben dem fuzzy-Suchoperator unterstützt unser Algorithmus weitere fortgeschrittene Suchanfrageoperationen effizient, und ohne Quellcodeduplikation wie von dem Standardansatz benötigt.

In Kapitel 10 rekapitulieren wir unsere Hauptbeiträge sowie Erkenntnisse. Des Weiteren erörtern wir die noch offenen Probleme unserer Arbeit und diskutieren wichtige zukünftige Verbesserungen.

1. Introduction

1.1. Towards an Efficient and Fully Error-tolerant Search

In a traditional information system, the user is supposed to specify a complete query in advance in terms of unambiguous and exact keywords. However, the user often makes mistakes typing the query due to limited knowledge about the search keywords or the underlying corpus. Hence, in many search applications, he or she is forced to use a try-and-see approach to find the desired information. For example, imagine a user searching for publications by the author `HongJiang Zhang` who does not know the exact spelling of the author's name. Autocompletion has been a first step towards alleviating this problem. With every letter being typed, the user receives instant results that may guide the search. A step further is a search that is tolerant to typing errors. In the following, we will take a closer look at two variants of this emerging trend in information search and discuss the state of the art as well as the current challenges.

1.1.1. Error-tolerant (fuzzy) Keyword-based Search

Query-side Tolerance: The reader might be already familiar with the fact that it is very desirable (and often necessary) that a search engine is robust against mistakes in the query. For example, when a user types `probablistic database`, the search engine should also deliver a ranked list of documents containing the correct keywords `probabilistic database`. Web search engines like Google provide this features by asking back "Did you mean: probabilistic database" or by suggesting this alternative query in a drop-down box. Search results for the alternative query are then either displayed proactively, in a separate panel, or by clicking on a corresponding link.

Document-side Tolerance: Assume the opposite, i.e., our keywords are correct, however, a relevant document in our corpus contains a misspelling for one of the keywords. Our information search would be still unsuccessful. This happens surprisingly often, either due to typos caused by the author of the document, or because of OCR errors. Web search engines do not provide this feature. The main reason is that in web search there are usually more than enough hits and the primary goal is to find the best such hits. However, in domain-specific search (for example, literature search), *recall* (fraction of relevant documents that are retrieved) is as important as *precision* (fraction of retrieved documents that are relevant) and sometimes even more important. For example, if there are only a dozen of papers about probabilistic databases in our corpus, we certainly do not want to miss the one with the misspelling `probablistic` in the title.

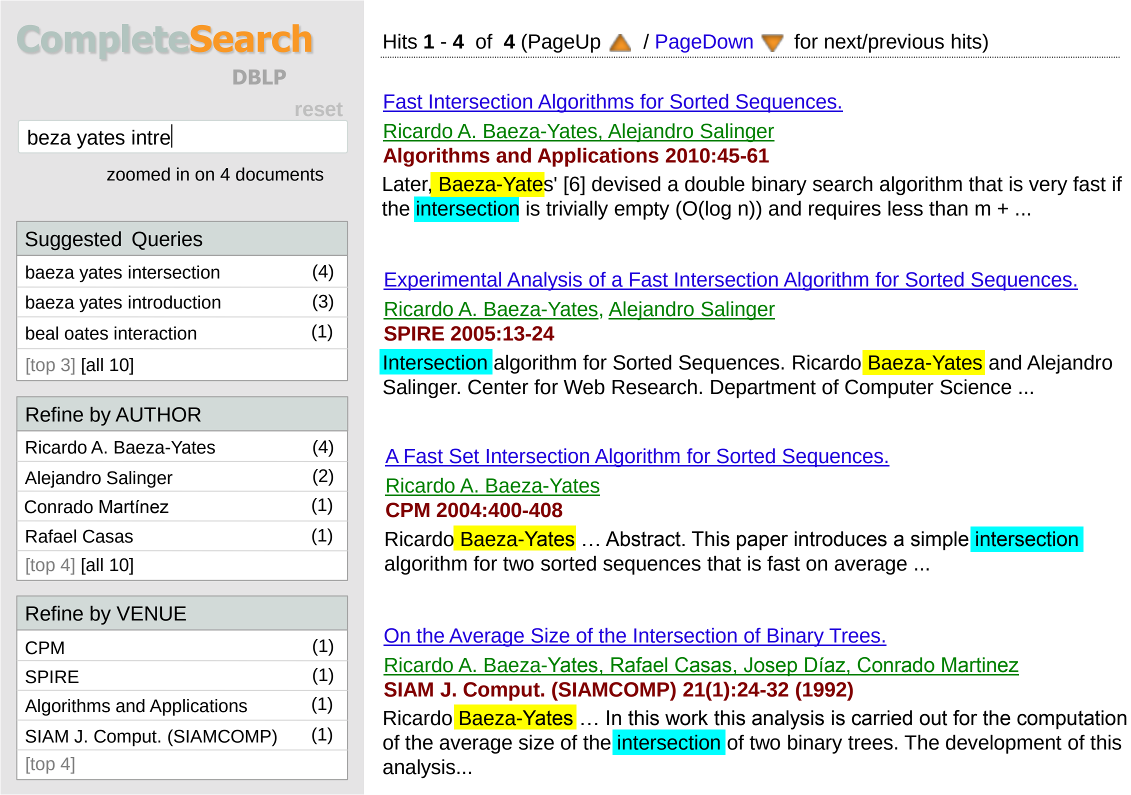
1.1.2. Error-tolerant (fuzzy) Autocompletion Search

Today, autocompletion has become a ubiquitous feature found to be useful in variety of search application. For example, all major search engines (including Google[1], Yahoo![2] and Bing[3]) support this feature at least to a certain extent. The error-tolerant variant of autocompletion search is another emerging trend in modern information search. In this search paradigm, with every letter being typed, we require an instant display of hits and completions containing the prefix approximately. For example, consider the name `Tchaikovsky`. It is likely that a user will already start typing an erroneous prefix of this keyword, for example `Tschai` or `Tczai`. On the other hand, the number of completions resulting from typing very short prefixes like `T` is likely to be too large to be useful.

[1] www.google.com

[2] search.yahoo.com

[3] www.bing.com



CompleteSearch
DBLP

reset

beza yates intre

zoomed in on 4 documents

Suggested Queries	
baeza yates intersection	(4)
baeza yates introduction	(3)
beal oates interaction	(1)
[top 3] [all 10]	

Refine by AUTHOR	
Ricardo A. Baeza-Yates	(4)
Alejandro Salinger	(2)
Conrado Martínez	(1)
Rafael Casas	(1)
[top 4] [all 10]	

Refine by VENUE	
CPM	(1)
SPIRE	(1)
Algorithms and Applications	(1)
SIAM J. Comput. (SIAMCOMP)	(1)
[top 4]	

Hits **1** - **4** of **4** (PageUp ▲ / PageDown ▼ for next/previous hits)

Fast Intersection Algorithms for Sorted Sequences.
Ricardo A. Baeza-Yates, Alejandro Salinger
Algorithms and Applications 2010:45-61
Later, Baeza-Yates' [6] devised a double binary search algorithm that is very fast if the intersection is trivially empty (O(log n)) and requires less than m + ...

Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences.
Ricardo A. Baeza-Yates, Alejandro Salinger
SPIRE 2005:13-24
Intersection algorithm for Sorted Sequences. Ricardo Baeza-Yates and Alejandro Salinger. Center for Web Research. Department of Computer Science ...

A Fast Set Intersection Algorithm for Sorted Sequences.
Ricardo A. Baeza-Yates
CPM 2004:400-408
Ricardo Baeza-Yates … Abstract. This paper introduces a simple intersection algorithm for two sorted sequences that is fast on average ...

On the Average Size of the Intersection of Binary Trees.
Ricardo A. Baeza-Yates, Rafael Casas, Josep Díaz, Conrado Martinez
SIAM J. Comput. (SIAMCOMP) 21(1):24-32 (1992)
Ricardo Baeza-Yates … In this work this analysis is carried out for the computation of the average size of the intersection of two binary trees. The development of this analysis...

Figure 1.1.: A current version of CompleteSearch doing fuzzy search with one of our new algorithms on DBLP. The query being typed is *beza yates intre* (the intended query is *baeza yates intersection*, referring to list intersection algorithms by the author Ricardo Baeza-Yates). It should be noted that Google currently does not provide any meaningful suggestion for this query.

1.1.3. Query Suggestion

It is desirable that a set of query suggestions along with the search results are shown to the user, especially if the number of hits is small. Consider the query `beza yates intre`. Among other, this query should result in documents that contain the keywords `baeza yates intersection`, referring to list intersection algorithms by the author Ricardo Baeza-Yates. Other good suggestions could be `baeza yates introduction`, `baeza yates interests`, but not for example, `baeza gates interval` since, although each word by itself is frequent, the whole query would lead to only few good hits. This example demonstrates the problem of computing the most likely *query suggestions*, which is a naturally accompanying problem to fuzzy search. Note that absence of query logs only exacerbates the problem since the suggestions must be computed by merely making use of the data in the corpus. Figure 1.1 shows a screenshot of our error-tolerant autocompletion search in action, searching the DBLP collection of computer-science articles[4] and responding to the query `beza yates intre`.

1.2. Challenges

Efficient index data structures that allow fast exact autocompletion search already exist. For the most prominent approaches the reader should refer to Bast and Weber [2006] and Ji *et al.* [2009]. However, there is surprisingly little work on indexes that provide efficient fuzzy keyword-based or fuzzy autocompletion search.[5] Google's web search in its current state offers fuzzy search in both variants as long as the autocompletion is on the last keyword and as long as the errors are on the side of the query. There is a fundamental difference in Google's approach, however. Namely, Google has the commodity of billions of queries. In particular, Google can suggest

[4] `dblp.uni-trier.de`, `www.dblp.org/search`

[5] Note that *approximate string matching*, which is about finding all approximate occurrences of a short pattern P in a long text T, is a better studied but a harder problem.

what the user has meant or intends to type based on hundreds of similar popular queries that have been already typed in. Hence, Google's "instant" (autocompletion) search and "Did you mean" query suggestion is based on *pre-compiled* list of queries. The underlying data structure still remains to be a (parallel) *inverted index*. As a consequence, if a query is not in the pre-compiled list (for example, an expert query with relatively narrow hit set), no instant search or suggestions will be provided to the user.

1.2.1. Absence of Query Logs

Snapping to popular queries from a pre-compiled list makes a lot of sense for web search, but is unsatisfactory for many vertical search applications (e.g. literature search, intranet search, e-mail search, desktop search, etc). This is because large query logs are rarely available and "expert queries" with small hit sets are the rule rather than the exception. In our approach we assume complete absence of query logs.

1.2.2. Large Datasets

Efficiency becomes a grand challenge in our problem setting when we face large collections (say more than 100 GB) on a single machine. Providing interactive query times (below 100 ms) for the above fuzzy search variants becomes a surprisingly hard problem. The main reason is that the total number of occurrences in a corpus similar to a given keyword or prefix increases faster than linearly with the size of the corpus. This is because the size of the vocabulary does not remain constant but rather increases with the size of the corpus according to *Heap's law* [Heaps, 1978]. The Heap's law states that the number of distinct terms in a corpus is approximately square-root of the total number of occurrences. In practice, this results in a roughly one order of magnitude larger volume to be dealt with by the system compared to exact search. As a result, scalability becomes a problem for algorithms that are rather efficient on comparatively smaller collections. The requirement for interactive query times only aggravates this problem. For example, according to our experiments, the recent *ForwardList* algorithm from Li *et al.* [2011] is reasonably fast on relatively small text collections but fails to achieve competitive running times on large text collections. This calls for algorithms and data structures that scale well on large data.

1.2.3. Engineering Challenges

Our work involves major engineering and technical challenges that require a careful *algorithm engineering*. Our first goal was algorithms that are efficient not only in theory, but also in practice. Our second goal was the implementation of a fully-fledged fuzzy search system. This involves engineering challenges different than that of an "ordinary" search system. For example, when fuzzy (prefix) search is employed, a matching document may potentially contain not only one but hundreds of similar, however, *non-literally* matching words that must be *highlighted* and shown to the user in form of *textual snippets*. When complex query operators are involved, each operator must be implemented twice, once on the index side, and once on the document (snippet) side. Hence, an important engineering aspect is how to avoid code duplication. Another engineering challenge is the *construction* of an index that supports fuzzy search. Important issues to be considered here are the index size and the construction efficiency.

1.2.4. Beyond Textual Similarity

There are instances of the fuzzy search problem that cannot be captured by the conventional textual similarity of words. One such instance are words that are only phonetically similar (e.g. `lynx` and `lincks`), however, their textual similarity (e.g. Levenshtein distance) is higher than a reasonable threshold. Another instance is the *word-boundary problem*. This happens when certain phrases are sometimes written as a single word and sometimes as multiple words (e.g. `dishwasher` and `dish washer`). It is especially relevant for languages with compound nouns such as German. The problem becomes harder when the phrase contains spelling mistakes (e.g. `tunder storm` instead of `thunderstorm`).

1.3. Contributions and Outline

The contribution of this dissertation is twofold. The first part deals with efficient data structures and algorithms that are the core of our fuzzy search. The second part addresses important technical aspects of an error-tolerant search engine. All of our algorithms and data structures have been integrated with the CompleteSearch engine [Bast and Weber, 2006], which offers results and suggestions as you type.

We start by formalizing the main problems considered in this dissertation. We then present our algorithms for the fuzzy word matching problem (also known as approximate dictionary searching). The fuzzy word matching problem is about efficiently computing all words in a large dictionary similar to a given word with respect to the *Levenshtein distance* [Levenshtein, 1966]. The Levenshtein distance is defined as the minimum number of edit operations (insertions, deletions and substitutions) needed to transform one string into another. We then present our algorithms for the fuzzy prefix matching problem (also known as error-tolerant autocompletion). Analogously, the fuzzy prefix matching problem is about computing all words in a large dictionary similar to a given prefix with respect to the *prefix Levenshtein distance*. The prefix Levenshtein distance is defined as the minimum Levenshtein distance between a prefix and any prefix of a given word. These two problems are prerequisites for the central problem considered in this dissertation, namely, efficient fuzzy keyword-based and autocompletion search. The fuzzy keyword-based (autocompletion) search problem is about efficiently computing a ranked list of documents that contain the query keywords approximately or exactly with respect to the (prefix) Levenshtein distance, and simultaneously computing a ranked list of query suggestions. The final two problems that we consider are efficient construction of an index that supports fuzzy search and efficient snippet generation (also known as document summarization) for fuzzy search.

In the following, we provide a more detailed chapter-by-chapter breakdown for each of the above (sub)problems, with a short summary of the contribution.

In Chapter 2, we formalize the algorithmic problems considered in this thesis. In addition, we present an ad-hoc solution to our central problem by using a standard inverted index. We will consider this the baseline algorithm throughout this dissertation. In fact, many open-source search engines, such as Lucene[6], employ this approach to answer fuzzy search queries. Besides the inverted index, we consider other alternative data structures for fuzzy search.

In Chapter 3, we present two practical algorithms for the fuzzy word matching problem with two different trade-offs. Our first algorithm is particularly efficient on short words. It is based on a technique called *truncated deletion neighborhoods* that allows a practical index which retains most of the efficiency of deletion neighborhood-based algorithms for dictionary search. Our second algorithm is particularly efficient on longer words. It makes use of a signature based on the *longest common substring* between two words. Instead of q-gram indexes, our algorithm is based on permuted lexicons, providing access to the dictionary via cyclic substrings of arbitrary lengths that can be computed in constant time. Our algorithms, depending on the threshold, improve the previously best algorithms [Schulz and Mihov, 2002; Li *et al.*, 2008] for up to one order of magnitude. Parts of this work have been published in the ACM Symposium of Applied Computing (SAC 2009) [Celikik and Bast, 2009b]. Parts of the work in this and the next three chapters are currently under re-review by the ACM Transaction of Information Systems (TOIS) journal.

In Chapter 4, we show how to extend the algorithms from the previous chapter to achieve efficient (off-line) fuzzy prefix matching. In addition, we provide a simple incremental (on-line) algorithm that is sufficiently fast in practice and even competitive to the state-of-the-art algorithm from Ji *et al.* [2009] when the prefix is sufficiently long.

In Chapter 5, we propose two novel data structures called fuzzy word and fuzzy prefix index and a new query processing algorithm. Our algorithms provide fuzzy full-text search that is (1) robust against errors on the side of the query, (2) robust against errors on the side of the documents, (3) supports both keyword-based and autocompletion search, (4) does not rely on a pre-compiled list of queries, and (5) it has interactive query times also for large text collections (up to 100 GB in size) on a standard machine. Our algorithms improve the state of the art in both settings, when the index resides in memory (for up to one order of magnitude) and when the index resides on disk (for up to a factor of 4). Parts of this work have been published in the 2nd International Workshop on Keyword Search on Structured Data (KEYS 2010) [Bast and Celikik, 2010].

In Chapter 6, we present our query suggestion mechanism. First, we propose an algorithm for computing a

[6] lucene.apache.org

ranked list of query suggestions in the absence of query logs. Our algorithm makes use of the result lists already (partly) computed from our fuzzy index and requires only a small fraction of the total query processing time. Second, we present a time and space efficient method for precomputing and indexing popular phrases from the searched collection that are shown to the user as completions/predictions of the query being typed.

In Chapter 7, we shed light on relevant extensions and details related to our fuzzy search that have not been discussed in the previous chapters. This includes phonetic errors, more sophisticated string distances, dealing with word-boundary errors, ranking and top-*k* processing and support for different character encodings.

In Chapter 8, we deal with an important technical aspect when it comes to search engines, namely efficient index construction. We individually consider the scenarios when the autocompletion is exact and when the autocompletion is fuzzy. We show that our indexes can be constructed at least as fast as the inverted index and often twice as fast. This is remarkable in two respects. First, because our indexes are practical but offer much more than an ordinary inverted index, and second, because fast construction of the inverted index has been the subject of extensive research and is well understood, leaving little room for further improvement of the state of the art. Parts of this work have been published in the 16th International Symposium on String Processing and Information Retrieval (SPIRE 2009) [Celikik and Bast, 2009a] and in the ACM Transaction of Information Systems (TOIS) journal (Vol 29, No. 3) [Bast and Celikik, 2011].

In Chapter 9, we consider another important technical aspect, namely result-snippet generation and result-snippet caching. Snippet generation is usually implemented by searching, at query time, for occurrences of the query words in the top-ranked documents. As already mentioned, this becomes problematic when the matches are fuzzy, i.e., non-literal. Another problem is code duplication when complex query operators are involved. Yet another problem is generating snippets for large documents. We present a novel and universal snippet generation algorithm that addresses all three problems. Our new approach localizes the snippets solely based on the index and achieves a higher cache hit-ratio compared to traditional approaches in snippet generation which is key to high performance. This work has been submitted for publication to the ACM Transaction of Information Systems (TOIS) journal.

In Chapter 10, we recapitulate our main contributions as well as lessons learned. We also shed light on the loose ends of our work and discuss and propose possible future improvements.

2. Basic Problem Definition and Baseline Algorithm

In this chapter, we formalize the central problems considered in this dissertation and show how the inverted index, the standard data structure in information retrieval, is used to solve them.

In the following Section 2.1, we bring in a small but important set of necessary terminology that we will use throughout most of the dissertation. In Section 2.2, we define our family of problems formally and then in Section 2.3 give further insights into the problems by providing straightforward but concrete solutions that will serve as a baseline.

2.1. Preliminaries

Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of documents and let $W = \{w_1, \ldots, w_m\}$ be its dictionary, i.e., the set of all distinct words that appear in D. For the sake of consistent terminology, we will refer to strings as words. We denote the length-n prefix of a word w as $w[n]$, where $w[n] = w$ if $n > |w|$. To denote that w_1 is a prefix of w_2, we will use $w_1 \preceq w_2$. $\text{LD}(q, w)$ will denote the Levenshtein distance [Levenshtein, 1966] between a keyword q and a word $w \in W$ and δ will denote the distance threshold. If not otherwise specified, we will assume that δ is a function of the keyword length, defined as

$$\delta(n) = \begin{cases} 1 & \text{if } n \leq 5 \\ 2 & \text{if } 5 < n \leq 10 \\ 3 & \text{otherwise} \end{cases}$$

This is because we would like to allow more error tolerance on long keywords and less error tolerance on shorter keywords. We define LD separately for words and prefixes as follows.

Definition 2.1.1 (Word Levenshtein Distance)**.** *Given two words w_1 and w_2, the word Levenshtein distance (denoted as* WLD*) is simply the Levenshtein distance between w_1 and w_2 defined as the minimum number of edit operations (insertions, deletions, substitutions) required to transform w_1 into w_2.*

For example, the word Levenshtein distance between `smith` and `smyth` is 1. The word Levenshtein distance can be computed by a well known dynamic-programming algorithm in $O(|w_1| \cdot |w_2|)$ time and $O(min\{|w_1|, |w_2|\})$ space. The earliest reference to this algorithm dates back to Vintsyuk [1968], but has later been rediscovered by various authors in various areas, including Needleman and Wunsch [1970], Sankoff [1972], Sellers [1974] and others. Although slow, the dynamic programming algorithm is very flexible in terms of adapting various distance functions. Moreover, it is easy to generalize the recurrence to handle substring substitutions [Ukkonen, 1983]. There are number of solutions that improve this algorithm for decreased flexibility. They are typically based on properties of the dynamic programming matrix, such as traversal of automata, bit-parallelism and filtering. For a good survey the interested reader should refer to Navarro *et al.* [2000a].

Definition 2.1.2 (Prefix Levenshtein Distance)**.** *Given a prefix p and a word w, the prefix Levenshtein distance (denoted as* PLD*) between p and w is defined as the minimum word Levenshtein distance between p and a prefix of w.*

A similar notion of prefix Levenshtein distance (called "extension distance") has already been introduced in Chaudhuri and Kaushik [2009]. For example, the prefix Levenshtein distance between `algro` and `algorithmic` is 1, because the word Levenshtein distance between `algro` and `algo` is 1 (and there is no other prefix of smaller word Levenshtein distance to `algro`). Note that unlike the word Levenshtein distance, the prefix Levenshtein distance is not commutative. Whenever $\text{PLD}(p, w) \leq \delta$, we will informally say that w is a *fuzzy completion* of p. Figure 2.1 illustrates the dynamic programming matrix for the strings `algro` and `algorithm`. The prefix Levenshtein distance simply corresponds to the minimum value in the last row of the matrix. The dynamic programming algorithm for the word Levenshtein distance can be easily adapted to compute the prefix Levenshtein

Table 2.1.: Dynamic programming table for the strings *algorithm* and *algro*. Only the gray cells should be considered when computing the prefix Levenshtein distance when the threshold is 1.

	ϵ	**a**	**l**	**g**	**o**	**r**	**i**	**t**	**h**	**m**
ϵ	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
l	2	1	0	1	2	3	4	5	6	7
g	3	2	1	0	1	2	3	4	5	6
r	4	3	2	1	1	1	2	3	4	5
o	5	4	3	2	1	2	2	3	4	5

distance in time $O(\delta \cdot |w|)$ as follows: fill only the cells that are at most δ cells away from the main diagonal (those in gray color) and treat all other cells as if ∞ were stored.

2.2. Problem Definitions

Definition 2.2.1 (Fuzzy word / autocompletion matching)**.** *Given a query q, a threshold δ, and a dictionary of words W, the fuzzy word / autocompletion matching problem is to efficiently find all words $w \in W$, such that* $\mathrm{LD}(q, w) \leq \delta$*, where* LD *is the word / prefix Levenshtein distance.*

We will first present our algorithms for the fuzzy word matching problem and then show how to extend them to fuzzy autocompletion (prefix) matching. Our algorithms for the fuzzy word matching problem are presented in Chapter 3. Efficient fuzzy autocompletion matching is dealt with in Chapter 4.

Recall that our fuzzy search involves computing a set of query suggestions for a given query Q in the absence of any external information like query logs. This makes the problem significantly more challenging; see Chapter 6 for details. A vital information that we will make use of for the relevancy of a query suggestion Q' instead, is how often Q' occurs in our corpus D and in what documents. For the sake of clarity, in this section we will introduce a simplified version of this information and refer to it as the *co-occurring frequency* of Q'.

Definition 2.2.2 (Co-occurring frequency)**.** *The co-occurring frequency of an n-tuple of words Q' in a set of documents D is the total number of documents $d \in D$ that match Q' exactly. A document $d \in D$ matches Q' exactly if all words in Q' are contained in d.*

Besides the co-occurring frequency, we will make use of other measures for the quality of a given query suggestion. The exact scoring mechanism will be explained in finer detail in Chapter 6. We are now ready to define our central problem.

Definition 2.2.3 (Fuzzy keyword-based / autocompletion search)**.** *Given a set of documents D, its dictionary W, a query $Q = \{q_1, \ldots, q_l\}$ and a threshold δ, let $Q_i = \{w \in W \mid \mathrm{LD}(q_i, w) \leq \delta\}$ for $i = 1 \ldots l$ and let $\mathcal{Q} = Q_1 \times Q_2 \times \ldots \times Q_l$, where* LD *is the Levenshtein word / prefix distance. The conjunctive fuzzy keyword-based / autocompletion search problem is to efficiently compute a pair (D', S), where $D' = \{d \mid d \in D, 1 \leq \forall i \leq l, \exists q_i' \in Q_i \text{ such that } q_i' \in d\}$ is the set of matching documents ranked by their relevance to Q, and $S \subseteq \mathcal{Q}$ is the set of top-k suggestions for Q, ranked by their co-occurring frequency in D'. The disjunctive fuzzy keyword-based / autocompletion search problem is to efficiently compute the set of matching documents $D' = \{d \mid d \in D, \exists q_i' \in Q_i \text{ such that } q_i' \in d\}$ ranked by their relevance to Q.*

If not otherwise stated, by *fuzzy search* we will implicitly refer to *conjunctive fuzzy search*. We will not associate a set S of fuzzy suggestions to D' for disjunctive fuzzy search since, in this case, S is not semantically well defined. Our algorithms for fuzzy keyword-based search and its autocompletion variant are presented in Chapter 5.

Example 2.2.4. *Assume we are in fuzzy prefix search mode and the user types the conjunctive query* `probab ases`*. Then we would like an instant display of the top-ranked documents that contain fuzzy completions of both* `probab` *and* `ases` *as well as the best suggestions for the intended full query in a separate box, for example* `probabilistic assessment`, `probability assessment`, *and* `probabilistic association`*. But not, for example,* `probabilistic ages`, *assuming that, although* `ages` *by itself is a frequent word, the whole query leads to only few good hits.*

Remark 1: We made a distinction between fuzzy keyword-based search and fuzzy autocompletion search, since the first problem is easier to solve than the latter, and some applications may involve only the first problem. The reason for this complexity difference lies in the number of possible matches. The number of similar completions of a relatively short prefix is typically orders of magnitude larger than the number of words similar to a given word (tens of thousands versus a few hundreds on the English Wikipedia, for prefixes of lengths 4 to 7).

Remark 2: The reader may wonder why, in Definition 2.2.3, we seek to find the set D' of all documents matching *any* query similar to Q. For example, why, for our example query `probab ases` from above, would we want hits for `probabilistic assessment` interspersed with hits for the completely unrelated `probability ages`. An alternative definition would ask only for those documents matching the top-ranked query suggestion from S. We want to make three remarks here. First, with all the approaches we tried, the alternative problem is as hard to solve as the problem defined above, because to compute S we have to at least compute the set of matching documents D' for Q. Second, the queries from S are often related, like `probabilistic assessment` and `probability assessment` in the example above, and it does make sense to present combined results for these two. Third, with a proper ranking, the most relevant documents will be at the top of the list. If these happen to be for different queries from S this provides some potentially interesting diversity for the user. If these happen to be all for the same (top-ranked) query, there is no difference between the two definitions for this query.

2.3. Baseline Algorithm

In this section, we will describe an algorithm that uses an ordinary inverted index to address the fuzzy keyword-based and autocompletion search problem. We will assume that we already have an efficient solution for the fuzzy word and autocompletion matching problem (Chapters 3 and 4).

Given a query (consisting of one or more keywords), the first part of the fuzzy search problem defined above requires computing the list of matching documents D' containing these keywords or words similar to them. The second part requires computing the set of top ranked query suggestions S.

2.3.1. Computing the Set Of Matching Documents D'

In this section, we show how to solve the first part of the fuzzy search above by using an inverted index. Given a query $Q = (q_1, \ldots, q_l)$, recall that

$$Q_i = \{w \in W \mid \mathrm{LD}(q_i, w) \leq \delta\}$$

It is not hard to see that computing the list of matching documents D' for a conjunctive query Q, requires computing the following *intersection of union lists*

$$\left(\bigcup_{j=1}^{|Q_1|} L_{w_{1,j}}\right) \cap \left(\bigcup_{j=1}^{|Q_2|} L_{w_{2,j}}\right) \cap \ldots \cap \left(\bigcup_{j=1}^{|Q_l|} L_{w_{l,j}}\right) \tag{2.1}$$

where $L_{w_{i,j}}$ is the inverted list of $w_{i,j} \in Q_i$, for $j = 1, \ldots, |Q_i|$. Analogously, when Q is disjunctive, computing D' requires computing the corresponding *merging of union lists*, where each intersection operator in Equation 2.1 is replaced by the union (merge) operator.

Lemma 2.3.1. *By using an inverted index, the above intersection of union lists (analogously, merging of union lists) can be computed in time*

$$\sum_{i=1}^{l} (c_1 + c_2 \cdot \log |Q_i|) \cdot |L_{q_i}|$$

where c_1 is the constant factor in the running time of list intersection, c_2 is the constant factor in the running time of multi-way merge and $L_{q_i} = \cup_j L_{w_{i,j}}$.

Proof. The intersection of union list can be computed in three steps as follows. First, compute each Q_i for $i = 1, \ldots, l$ by using word-Levenshtein distance if in keyword-based mode or prefix Levenshtein distance if in autocompletion mode. Then compute L_{q_i} for $i = 1, \ldots, l$ by a multi-way merge of $L_{w_{i,j}}$, $j = 1, \ldots, |Q_i|$ in time $c_2 \cdot |L_{q_i}| \cdot \log |Q_i|$. At the end, compute $L_{q_1} \cap \ldots \cap L_{q_l}$ in time $c_1 \cdot \sum_{i=1}^{l} |L_{q_i}|$ by a simple linear intersection.[1] □

We will refer to this algorithm as `Baseline`. We highlight two problems of `Baseline`. First, `Baseline` is disk-inefficient. When the index resides on disk, reading many lists involves many disk seek operations.[2] This can make the algorithm prohibitive if the indexed corpus is large and/or the disk is slow. Second, `Baseline` is in-memory or computationally inefficient. Fully materializing each union list L_{q_i} is expensive since it involves merging a large number of lists with high total volume. In particular, $|Q_i|$ is typically in the order of hundreds for fuzzy word matching and in the order of thousands for fuzzy prefix matching. Moreover, each L_{q_i} is up to an order of magnitude larger than the corresponding inverted list of q_i. Hence, for each posting in L_{q_i} we have to spend $c_1 + c_2 \cdot \log |Q_i|$ time, where c_2 is relatively large.

2.3.2. Computing the Set Of Query Suggestions S

Let $\mathcal{Q} = Q_1 \times \ldots \times Q_l$ be the set of all candidate suggestions for Q. The algorithmic problem that we consider is to compute the co-occurring frequencies of the query suggestions from $\mathcal{Q}$ in D'. It is less obvious how to solve this problem reasonably by using an ordinary inverted index. One way (also suggested in Manning *et al.* [2008]) is to compute the intersections $L_{q'_1} \cap \ldots \cap L_{q'_l} \subseteq D'$, where $(q'_1, \ldots, q'_l) \in \mathcal{Q}$. However, this is feasible only if the set of reasonable suggestions has been already restricted. If all query suggestions from $\mathcal{Q}$ are considered, then the running time of this approach can become prohibitive as shown in the following lemma.

Lemma 2.3.2. *Given a query Q, the co-occurring frequencies of all query suggestion in $\mathcal{Q}$ by using an inverted index can be computed in time proportional to*

$$\sum_{i=1}^{l} \left(\prod_{j=1}^{i-1} |Q_j| \cdot |L_{q_i}| \right)$$

Proof. The co-occurrence frequency of Q' is equal to $|L_{q'_1} \cap \ldots \cap L_{q'_l}|$. Observe that to compute all l-tuples of such intersections, each posting in L_{q_i} must be touched $\prod_{j=1}^{i-1} |Q_j|$ times. □

If we assume, for simplicity, that all sets Q_i have sizes equal to m, the running time of the algorithm then is $O(m^{l-1} \cdot |L_{q_l}|)$. This becomes prohibitively large already for queries with three keywords.

[1] Thanks to its perfect locality of access and compact code, linear list intersection in our experiments was a faster option compared to other asymptotically more efficient intersection algorithms based on binary searches [Demaine *et al.*, 2000; Baeza-Yates, 2004] when the list sizes do not vary extremely.

[2] In practice, a disk seek is required only for inverted lists of words that are not contiguous in the dictionary. Note that lexicographic order of inverted lists is not always insured by the index construction algorithm [Heinz and Zobel, 2003a].

3. Efficient Fuzzy Word Matching

This chapter is about efficient fuzzy word matching, also known as approximate dictionary search. We combine various approaches from the literature to propose two practical and flexible algorithms with two different trade-offs that are suited for a dynamic distance threshold (introduced in the previous chapter). In addition, our first algorithm is particularly efficient (in time and space) when the words are short. Our second algorithm is particularly efficient when the words are long or when the distance threshold is low.

In the following Section 3.1, we give a short survey of the state of the art. Then in Sections 3.2 and 3.3 we present our new fuzzy word matching algorithms and test them in Section 3.4. As competitors we choose algorithms that are the fastest to our knowledge.

3.1. Related Work

There are numerous algorithms in the literature that could be reasonably applied to solve the fuzzy word matching problem (also known as approximate dictionary searching). In this work we consider offline or indexing algorithms based on Levenshtein distance that are efficient when the distance threshold is relatively small (e.g., not larger than 3 errors), when the length of the strings is relatively short (e.g., when the strings are words) and when their number is large (e.g., more than 5 million words).

Existing methods for fuzzy word matching can be categorized as follows:

- *q-gram filtering and pattern partitioning* methods [Willett and Angell, 1983; Jokinen and Ukkonen, 1991; Navarro, 2001; Chaudhuri *et al.*, 2006; Bayardo *et al.*, 2007; Li *et al.*, 2008; Xiao *et al.*, 2008b,a]
- Methods based on *neighborhood generation* [Mor and Fraenkel, 1982; Du and Chang, 1994; Myers, 1994; Russo *et al.*, 2009]
- Methods based on *tries* and *automata* [James and Partridge, 1973; Ukkonen, 1993; Baeza-Yates and Gonnet, 1999; Schulz and Mihov, 2002; Cole *et al.*, 2004; Mihov and Schulz, 2004]
- Methods based on *metric spaces* [Baeza-yates and Navarro, 1998; Chávez *et al.*, 2001; Shi and Mefford, 2005; Figueroa *et al.*, 2006]

A strict taxonomy is often inaccurate since some algorithms combine various approaches. In the following we provide a short summary for each category. A recent and extensive survey that addresses almost all aspects of the topic, both experimentally and theoretically, can be found in Boytsov [2011]. For more details the reader should refer there.

The *q-gram filtering approach* is by far the most common in the literature. Each string is represented as a set of q-grams. A q-gram is a substring with a fixed length of q characters. The basic algorithm converts the constraint given by the distance function into a weaker q-gram overlap constraint and finds all potential matches that share sufficiently many q-grams by using a q-gram index. The one problem for all of these approaches is the large number of visited strings (in the worst case all records with at least one q-gram in common). Therefore, various optimizations and filtering techniques are employed to minimize the number of visited strings. Typical optimizations include prefix filtering [Xiao *et al.*, 2008a,b; Bayardo *et al.*, 2007] and skipping [Li *et al.*, 2008]. A good representative (and readily available) algorithm for this category is `DevideSkip` from Li *et al.* [2008], implemented as a part of the Flamingo project on data cleaning.[1] The main idea is to skip visiting as many strings as possible while scanning the q-gram lists by exploiting various differences in the lists. The first optimization exploits the value differences of the string ids by using a heap. The second optimization exploits the differences among the list sizes such that the candidates in the longest lists are verified by using a binary search instead of a heap.

[1] http://flamingo.ics.uci.edu/

Neighborhood generation methods generate all possible strings obtainable by applying up to δ errors and then resort to exact matching of neighborhood members. The errors could be insertions, deletion or substitutions (full-neighborhood generation) or deletions only (deletion-neighborhood generation). To our knowledge, this class of algorithms in general is the fastest for our problem setting, however their exponential space complexity often makes them infeasible in practice. A similar observation has been made in Boytsov [2011]. A deletion-neighborhood based algorithm is covered in greater detail in Section 3.2.

A *prefix tree* or a trie is an ordered tree data structure used to store strings such that all the descendants of a node have a common prefix of the string associated with that node. Using a recursive trie traversal is a classical approach to compute the Levenshtein distance of a string against a dictionary of strings. The savings come from the fact that the distance is calculated simultaneously for all strings sharing a prefix. Pruning of the traversal takes place whenever the minimum value in the current column is larger than the threshold. One of the most prominent methods in this category has been proposed in Mihov and Schulz [2004]. It combines tries with pattern partitioning and neighborhood generation. The main contribution is an algorithm based on a pair of tries; one built over the dictionary words and another built over the reversed dictionary words. At query time, the pattern is split into two parts and series of $\delta + 1$ two-step subqueries are launched. The traversal of the tries is navigated by using a deterministic Levenshtein automaton. In Boytsov [2011], this algorithm is referred to as FB-tree.

Metric space approaches exploit the triangle inequality of the distance function to perform recursive partitioning of the space at index time by using specially selected elements from the dataset called *pivots*. The obtained partitioning is usually represented as a tree. In an earlier work [Celikik and Bast, 2009b], we have found out that these approaches are inferior (with respect to time or memory or both) when it comes to Levenshtein distance in our problem setting. Similarly, the compared metric-space-based algorithm in Boytsov [2011] is one of the slowest with very low filtering efficiency.

3.2. Algorithm: DeleteMatch

This section is about `DeleteMatch`, a practical algorithm for fuzzy word matching that is particularly efficient (in time and space) when the words are short. We first introduce the notion of an n-subsequence of a word w and explain its role as a signature (Section 3.2.1). In a nutshell, a subsequence of a word w is obtained by applying character deletions on a set of positions in w. The set of all subsequences of w is known as the deletion neighborhood of w. We describe a known method based on indexing the full deletion neighborhood of each word in the dictionary. We show that the resulting algorithm is fast but impractical due to its enormous index (Section 3.2.2). We then propose a novel indexing method called *truncated deletion neighborhoods*, which, when combined with compression, dramatically reduces the space usage of the algorithm. We show that our new method results in an algorithm with a practical index that retains most of the efficiency for dictionary search (Sections 3.2.3 and 3.2.4).

3.2.1. Deletion Neighborhoods and Subsequences

Given a word w, an n-subsequence of w for $n \leq |w|$ is the sequence of characters obtained by deleting any n characters from w. Let p be a l-tuple of delete positions in w and let $s(w, p)$ be the $|p|$-subsequence of w obtained by deleting the characters with positions in p. The following example shows how two words w_1 and w_2 with $\mathrm{WLD}(w_1, w_2) = 2$ share a long common subsequence.

Example 3.2.1. *Let* w_1=`algorythm` *and* w_2=`algoritm`*. Observe that* $s(w_1, (6, 8))$=$s(w_2, (6))$=`algortm`*, i.e., the words match on the subsequences corresponding to the l-tuples of delete positions* $(6, 8)$ *and* (6).

Unlike q-grams, n-subsequences retain much of the information of the original string. In the following we define the *n-deletion neighborhood* of a word w recursively, given some $0 < n \leq |w|$.

Definition 3.2.2. *The n-deletion neighborhood* $U_d(w, n)$ *of a word w consists of all k-subsequences of w, for* $k = 0 \ldots n$

$$U_d(w,n) = \begin{cases} w & \textit{if } n = 0 \\ \bigcup_{i=0}^{|w|} U_d\left(s(w,i), n-1\right) & \textit{otherwise} \end{cases}$$

It is intuitive that any two words within a distance threshold δ share a long common subsequence in their δ-deletion neighborhoods. The following lemma gives a more formal statement.

Lemma 3.2.3. *Given a threshold δ, let w_1 and w_2 be two words with* $\mathrm{WLD}(w_1, w_2) \leq \delta$. *Then there exist a subsequence $s \in U_d(w_1, \delta) \cap U_d(w_2, \delta)$ with $|s| \geq \max\{|w_1|, |w_2|\} - \delta$.*

Proof. The proof is based on constructing matching l-tuples of delete positions p_1 and p_2 in w_1 and w_2 such that $s(w_1, p_1) = s(w_2, p_2)$ by using the sequences of edit operations that transform w_1 to w_2 and w_2 to w_1. A detailed version is given in the appendix. □

3.2.2. The Mor-Fraenkel Method

Given a dictionary W, a threshold δ and a query word q, Lemma 3.2.3 immediately gives rise to an algorithm based on indexing subsequences. What follows is a summary of the Mor-Fraenkel method originally proposed in Mor and Fraenkel [1982] and Muth and Manber [1996]. The algorithm consists of an indexing and a searching procedure. The indexing procedure generates the full δ-deletion neighborhood of each $w \in W$ and stores all possible triples (s, p_w, c_w) for each subsequence s, where $s = s(w, p_w)$, p_w is an l-tuple of delete positions ($l \leq \delta$) and c_w is an l-tuple that stores the deleted characters. Each l-tuple is indexed by using s as a key. The original word w can be recovered by inserting characters from c_w in s at positions in p_w. The search procedure consists of generating all triples (s, p_q, c_q) from the deletion neighborhood of q. It is not hard to show that the Levenshtein distance between q and a word w such that $\exists s \in U_d(q, \delta) \cap U_d(w, \delta)$ can be efficiently computed as $|p_q| + |p_w| - |p_q \cap p_w|$.

The main drawback of this algorithm is its high space complexity. Given that each deletion-neighborhood entry requires $O(m)$ bytes in average, the space complexity is equal to

$$O\left(|W| \sum_{i=0}^{\delta} m \cdot \binom{m}{i}\right) = O\left(|W| \cdot m^{\delta+1}\right) \tag{3.1}$$

bytes, where m is the average word length. In practice, due to the many and long signatures, this algorithm has a prohibitive space demand. For example, for natural language dictionaries the index size is more than 100 times larger than the size of the dictionary for $\delta = 2$ [Boytsov, 2011].

There are existing methods for succinct representation of full δ-deletion dictionaries. Mihov and Schulz [2004] proposed to represent deletion neighborhoods for $\delta = 1$ in the form of minimal transducers. A transducer $T(s)$ for a dictionary W is a deterministic finite state automaton with an output. A minimal transducer is a transducer with minimal number of states. In summary, if $T(s)$ accepts a string s then $T(s)$ outputs all words $w \in W$ such that $s(w, p) = s$ for some p. This method has been shown to produce up to one order of magnitude smaller indexes for $\delta = 1$. However, it has not been verified in practice whether the indexes are smaller for $\delta > 1$ [Boytsov, 2011]. A similar method based on an equivalent list dictionary transducer for $\delta = 1$ has been proposed in Belazzougui [2009]. The practicality of this method has been not experimentally investigated.

3.2.3. Truncated Deletion Neighborhoods

We propose a method that avoids generating the full deletion neighborhood of each $w \in W$ at the price of a slightly increased verification cost on longer words. In addition, unlike related methods, our method uses standard data structures that run efficiently on today's hardware [Belazzougui, 2009].

What follows is a variant of the Mor-Fraenkel method. Instead of storing the auxiliary data for faster verification, for each indexed subsequence we store only its word id. The candidate matches are verified by using a fast bit-parallel version of the dynamic programming algorithm from Myers [1999].[2] The index procedure iterates over W and computes the inverted list $I(s)$ of each $s \in \bigcup_w U_d(w, \delta)$. The inverted list $I(s)$ of s is the sorted list of the word ids of all $w \in W$ with $s \in U_d(w, \delta)$, indexed by using s as a key. At query time, the search procedure obtains the inverted list $I(s)$ of each $s \in U_d(q, \delta)$ and for every $w \in I(s)$ verifies whether $\mathrm{WLD}(q, w) \leq \delta$. The seen words are marked to avoid duplicates as well as redundant computations.

[2] Despite this, the space usage remains prohibitive. For example, on a machine with 30 GB of RAM we could not index a clean version of the dictionary of a dump of the English Wikipedia with size of about 9 million distinct words, word length limit of 20 characters and a threshold $\delta = 2$.

Table 3.1.: Average number of distance computations per match for different query word lengths and different thresholds δ by using the simplified variant of the Mor-Fraenkel method from Section 3.2.3 on the dictionary of the DBLP collection with around 600K distinct words (the last three rows show the average number of matches per query length).

	4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
$\delta = 1$	2.2	2.0	1.5	1.3	1.2	1.2	1.2	1.2	1.2	1.2	1.2
$\delta = 2$	3.0	3.6	3.0	2.4	1.8	1.6	1.4	1.3	1.2	1.2	1.1
$\delta = 3$	2.2	3.5	4.2	3.8	3.3	2.6	2.1	2.0	1.5	1.5	1.1
$\delta = 1$	40	20	12	8	7	6	7	9	9	6	7
$\delta = 2$	917	395	154	66	31	20	21	17	21	20	12
$\delta = 3$	11,077	5,479	1,986	722	222	100	51	55	42	24	22

Table 3.2.: Size of the full and truncated deletion neighbrohoods built on the dictionary of the English Wikipedia (around 9 million words and 84 MB in size).

Prefix length	# Subsequences	Size
∞	1.4 billions	15 GB
8	67.1 millions	382 MB
7	17.0 millions	84 MB
6	2.6 millions	11 MB
5	0.9 millions	3 MB

The above algorithm can be regarded as filtering algorithm. It remains fast in practice due to the long and discriminative signatures that result in relatively short inverted lists. Table 3.1 shows the average number of distance computations per match for different query lengths. In average it performs between 1 and 3 distance computations per match overall. Hence, in average, it is not far from the "optimal filtering algorithm" that would perform a single distance computation per computed match.

The size of the index consists of two components, the size of the subsequence dictionary and the total size of the inverted lists. The size of the subsequence dictionary is proportional to the sum of the lengths of all subsequences. Since each inverted list contains at least one word id, the sizes of these two is usually similar.

Given a fixed $k > 0$, the *k-truncated version* of W is the dictionary W^k obtained by truncating each word w with $|w| > k$ to its k-prefix. In addition, W^k contains all words $w \in W$ with $|w| < k$. We assume that W is given in lexicographically sorted order. For each truncated word $w[k]$, we keep a pointer to the range $\{w \in W \mid w[k] \preceq w\}$ by storing the word id of the first word and the number of words in the range. Hence, we represent each range with a single integer (range-id). The following lemma will allow us to index the k-truncated dictionary W^k instead of the full dictionary W.

Lemma 3.2.4. *Let w_1 and w_2 be two words with* $\mathrm{WLD}(w_1, w_2) \leq \delta$ *and let $w_1[k]$ and $w_2[k]$ be their k-prefixes for $k \leq$* $\max\{|w_1|, |w_2|\}$. *Then $\exists s \in U_d(w_1[k], \delta) \cap U_d(w_2[k], \delta)$ with $|s| \geq k - \delta$.*

Proof. When $\mathrm{WLD}(w_1[k], w_2[k]) \leq \delta$ the statement obviously holds true due to Lemma 3.2.3. How to find a matching subsequence when $\mathrm{WLD}(w_1[k], w_2[k]) > \delta$ is shown in the appendix. □

The lemma requires truncating q at query time whenever $|q| > k$. The inverted lists now contain prefix ids. Since we index only short strings, the size of the index does not depend on the long words in W (e.g., outliers) that can have huge deletion neighborhoods and hence large impact on the index size. To reduce its size further, we will make use of another property of deletion neighborhoods.

Lemma 3.2.5. *Let w_1 and w_2 be two words with $|w_1| = |w_2|$ and* $\mathrm{WLD}(w_1, w_2) \leq \delta$. *Then $\exists s \in U_d(w_1, \delta) \cap U_d(w_2, \delta)$ with $|s| = |w_1| - \delta$ such that the number of deletions in w_1 and the number of deletions in w_2 are equal.*

Proof. According to Lemma 3.2.3 we already know that $\exists s \in U_d(w_1, \delta) \cap U_d(w_2, \delta)$ with $|s| \geq \max\{|w_1|, |w_2|\} - \delta$. Let p_1 be the l-tuple of delete positions in w_1 and p_2 be the l-tuple of delete position in w_2. Obviously, for each delete position in p_1 there must be a corresponding delete position in p_2 as otherwise the length of the resulting subsequences will not be equal. □

Table 3.3.: Index size and average running time (given in microseconds) of *DeleteMatch* for $\delta = 2$ and different truncation lengths (space/time trade-offs) on the dictionary of the DBLP collection with around 600K distinct words (the average number of matches was 241 words).

Prefix Length (k)	Index Size	Running Time		Distance Comp.	
		with filter	without filter	with filter	without filter
∞	755 MB	-	156 us	-	785
7	25 MB	185 us	272 us	855	1050
6	10 MB	316 us	661 us	1110	2310
5	6 MB	703 us	2637 us	1800	9747

Table 3.4.: Average running time per query length by indexing full (first row) and truncated deletion neighborhoods (second row).

4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
.4 ms	.3 ms	.1 ms	.06 ms	.04 ms	.04 ms	.04 ms	.05 ms	.06 ms	.06 ms
.4 ms	.3 ms	.1 ms	.06 ms	.07 ms	.07 ms	.06 ms	.07 ms	.06 ms	.06 ms

Recall that our dictionary consists mainly of k-prefixes. The above lemma allows us to index only subsequences of length $k - \delta$ instead of the full deletion neighborhoods. It is not hard to see that in the same time the efficiency of our algorithm is slightly improved because many subsequences are not generated and scanning their inverted lists is omitted.

We refer to the resulting set of subsequences as the *truncated deletion neighborhood* of W. It should be noted that if W contains many words with equal prefixes (e.g. in languages with compound nouns like German), then we could simply index the truncated deletion neighborhood on the reversed dictionary due to the property $\text{WLD}(w_1, w_2) = \text{WLD}(rev(w_1), rev(w_2))$, where $rev(w)$ is the reversed string w. The size of the subsequence dictionary of the truncated deletion neighborhood in practice is small, typically a fraction of W. Table 3.2 shows the size of the full and the truncated deletion neighborhoods for different values of k. The main observation is that the size of the full deletion neighborhood is almost 200 times larger than the size of the original dictionary, while already for $k = 7$ the size of the truncated deletion neighbrorhood is close to that of the dictionary. This allows us storing it in a conventional data structure for fast access such as hash table or a trie.

Note the total size of the inverted lists is reduced as well since they are significantly less in number and since ranges of words with a common k-prefix are represented by a single id. They are, however, significantly longer. To represent them efficiently, we employ the observation that k-prefixes in close neighborhoods in W^k with a subsequence in common have small gap values when the lists are gap encoded. The following is an example of a gap encoded inverted list for the subsequence "gffe" (the corresponding k-prefixes are shown only for convenience).

100563	1	1	1	1	1	3	2	135
gaffe	**gaffeo**	**gaffer**	**gaffet**	**gaffey**	**gaffie**	**gaffke**	**gaffne**	**gaiffe**

The sizes of the gaps depend on the common prefix lengths. For example, strings with common prefixes of length $k - 1$ have gap values of 1. The gaps are efficiently represented by variable-byte encoding [D'Amore and Mah, 1985]. This is a well known coding scheme that allows fast decoding for a slight loss of compression efficacy compared to bit aligned schemes [Scholer *et al.*, 2002]. For example, decoding 100,000 integers requires less than a millisecond. Hence, list decompression at query time takes only a small fraction of the total running time.

The truncation length parameter k allows a trade-off between the index size and the running time of the algorithm (see Table 3.3). Truncated deletion neighborhoods with $k = 7$ and $\delta \leq 3$ resulted in an index with size within a factor of 3 from the size of the dictionary with almost identical average running time. Setting k to 6 resulted in index with size less than the size of the dictionary at the price of twice higher average running time.[3] Instead of using a fixed k, a better strategy in practice is choosing a larger k on longer words and a

[3] This experiment was performed on the English Wikipedia with around 9M words and average word length of 9.3 characters; for more information see Section 3.4.

Table 3.5.: Average number of candidates per query when using the length and the q-gram count filter on suffixes for $q = 1$ and $q = 2$ (on the dictionary of the DBLP collection with around 600K distinct words).

Threshold	$q = 2$	$q = 1$
$\delta = 1$	29	33
$\delta = 2$	894	912
$\delta = 3$	11,298	11,255

smaller k on shorter words. We did not experiment extensively with this heuristic.

3.2.4. Suffix Filtering

Since the suffixes of the words are ignored, word truncation involves more false positive candidates and hence more distance computations. Therefore, additional filtering on the suffixes of the candidate matching words is required.

The *q-gram overlap (count) filter* and the *length filter* are two standard filters in the approximate string matching literature. The q-gram overlap filter [Sutinen and Tarhio, 1996a] mandates that the number of common (positional) q-grams must be at least $\max\{|w_1|, |w_2|\} - q + 1 - q \cdot \delta$. The length filter simply mandates $||w_1| - |w_2|| \leq \delta$. The challenge is to compute these filters efficiently in the absence of a q-gram index. This is because the suffixes are not indexed as it is undesirable to store any additional data that might increase the space usage of the algorithm. Moreover, we can afford only cheap filters since the suffixes are short and larger computational overhead can outweigh the pruning power.

The effectiveness of the q-gram overlap filter in general depends on the value of q. Similarly as in Gravano *et al.* [2001], we have determined that $q = 2$ gives the highest pruning power when no other filters are employed. Our suffix filtering is based on the observation that when the q-gram overlap filter is combined with truncated deletion neighborhoods, there was virtually no difference in the effectiveness of the filter between $q = 1$ and $q = 2$. To provide some evidence, Table 3.5 shows the average number of to-be-verified candidates after applying this filter for $q = 1$ and $q = 2$ on the suffixes of the candidate words.

The q-gram overlap filter for $q = 1$ is a weak version (lower bound) of the *unigram frequency distance* [Boytsov, 2011]. The unigram frequency distance of w_1 and w_2 is defined as the maximum between the number of unigrams in w_1 that are not in w_2 and the number of unigrams in w_2 that are not in w_1. It is well known that the unigram frequency distance lower bounds the Levenshtein distance. It can be efficiently computed by using *unigram frequency vectors*. A unigram frequency vector of a word w is a vector of size $|\Sigma|$ where $F_w[i]$ contains the number of occurrences of the character $\sigma_i \in \Sigma$ in w [Kahveci and Singh, 2001]. The following lemma will make use of the fact that the frequency distance has been already "computed" on the k-prefixes of the candidate matching words.

Lemma 3.2.6. *Assume* WLD$(q, w) \leq \delta$ *and that the k-prefix of w shares a subsequence of length $k - \delta \leq l \leq k$ with the k-prefix of q. Then the suffixes of w and q that start at position* $k+1$ *must share at least* $\max\{|w|, |q|\} - k - \delta$ *unigrams.*

Proof. We can safely overestimate the number of common unigrams between the k-prefixes of q and w by assuming $l = k$. Then it is trivial that their suffixes must share at least $\max\{|w|, |q|\} - k - \delta$ unigrams. □

The next lemma will allow us early termination of the computation of the frequency distance.

Lemma 3.2.7. *If* WLD$(q, w) \leq \delta$, *then the number of common unigrams between q and the prefix $w[i]$ of w must be at least $i - \delta$ for $i = \delta \ldots |w|$.*

Proof. The proof follows from the observation that if $w[i]$ contains $n > \delta$ unigrams that are not in q then any $w[j]$, $j \geq i$ will contain the same n unigrams that are not in q. □

Note that the same argument can be used to show that the lemma is valid for q-grams of any length.

Let *count* be the number common unigrams between q and w computed so far and let F_q be the frequency vector of q. We combine the above observations in a simple filter as follows:

1. Initially compute F_q and set *count* to k;

2. For $i = k + 1 \ldots |w|$, increase *count* by 1 if $F_q[w[i]] > 0$ and decrease $F_q[w[i]]$ by 1;

3. If the current value of *count* is below $i - \delta$, terminate the loop and conclude that $\text{WLD}(w, q) > \delta$;

4. If the final value of $count < \max\{|q|, |w|\} - \delta$, conclude that $\text{WLD}(w, q) > \delta$;

5. Restore F_q to its previous state by increasing $F_q[w[i]]$ by 1 for $i = k + 1 \ldots j$, where j is the last value of i in the loop in step 2.

3.3. Algorithm: PermuteScan

This subsection is about `PermuteScan`, an indexing method based on sequence filtering that is particularly efficient when the words are long relatively to the distance threshold or when the distance threshold is 1. Our algorithm combines two signatures: a novel signatures based on the notion of longest common substring shared by two words, and an already well known partitioning-based signature (Section 3.3.1). We show that this approach is more efficient than other sequence filtering methods based on q-grams. We first formulate the problem as an exact substring matching problem and employ an algorithm based on permuted lexicons that utilizes both signatures (Section 3.3.2). We then discuss which problem instances are hard for this algorithm and propose further optimizations to improve its running time (Sections 3.3.3, 3.3.4 and 3.3.5).

3.3.1. The Longest Common Substring Signature

The main observation in this section is that if $\text{WLD}(w_1, w_2) \le \delta$, then w_1 and w_2 must share at least one “long“ substring. Note that the substrings to this end are considered cyclic, i.e, a substring at the end of a word may continue at the beginning of the word. For example, `thmalg` is a substring of `algorithm`. We will refer to the length of this substring as *longest-common-substring signature*. It is formally stated as follows

Lemma 3.3.1. *If* $\text{WLD}(q, w) \le \delta$*, then q and w must share a substring of length at least* $\lceil \max\{|q|, |w|\}/\delta \rceil - 1$.

Proof. Without loss of generality, we can assume that the longer of the two words have length n and that the shorter word is obtained by performing δ deletions over the longer word. Now we would like to divide the longer word with the δ deletions into δ pieces such that the maximum length of a piece is minimized. This is achieved when each piece (except possibly the last) has equal length. Hence, the length of the first $\delta - 1$ pieces is equal to $\lceil n/\delta \rceil$. But since each of the δ deletions destroys a single character in each piece, the first $\delta - 1$ corresponding substrings between the two words will have length equal to $\lceil n/\delta \rceil - 1$. □

The following property is a direct consequence of Lemma 3.3.1

Property 3.3.2. *If* $\text{WLD}(q, w) \le \delta$*, then q and w must share a substring of length at least* $\lceil |q|/\delta \rceil - 1$ *(regardless of the length of w).*

This property is related to the following well known observation based on partitioning the query pattern [Wu and Manber, 1992].

Property 3.3.3. *Let* $\text{WLD}(q, w) \le \delta$*, where* δ *is integer. If q is partitioned into* $\delta + 1$ *pieces, then at least one of the pieces is an exact substring of w.*

In the original proposal q should be split into approximately equal pieces so that none of the pieces is too short.[4] One obvious difference to Lemma 3.3.1 is that for $\delta = 1$, Lemma 3.3.1 requires a common substring of length at least $|q| - 1$, whereas Property 3.3.3 requires a common substring of length at least $\frac{|q|}{2}$. This makes Lemma 3.3.2 more effective for $\delta = 1$. For many instances Lemma 3.3.1 requires a longer common substring because q and w can be considered as cyclic strings. Table 3.6 compares the differences between the substring lengths for different lengths of q. For example, if $|q| = 7$ and $\delta = 2$, then Lemma 3.3.1 requires a common

[4] We have also experimented with approaches for optimal partitioning of q similar to that from Navarro and Salmela [2009] based on substring frequencies, however, without significant improvements in the efficiency.

Table 3.6.: Differences between the required common substring lengths between a query q and a word w for varying lengths dictated by Property 3.3.2 (longest common substring) and Property 3.3.3 (pattern partitioning). 0 means that both properties require equal common substring lengths.

$\lvert q\rvert$	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$\delta = 1$	1	2	2	3	3	4	4	5	5	6	6	7
$\delta = 2$	0	1	0	1	1	1	1	2	1	2	2	2
$\delta = 3$	0	0	0	1	0	0	1	1	0	1	1	1

substring of length at least $\lceil \frac{7}{2} \rceil - 1 = 3$, while Property 3.3.2 a common substring of length at least $\lfloor \frac{7}{2+1} \rfloor = 2$. However, there are instances for which both properties require common strings of equal length. For these instances the longest common substring signature arising from Lemma 3.3.1 is wasteful since the common substrings can start at any position. In contrast, the substrings in Property 3.3.3 can start only at $\delta + 1$ fixed positions in q. The former requires checking more substrings and hence more work.

Therefore, we choose between the two signatures based on each particular instance. We opt for a simple criteria as follows. If the longest common substring signature requires a longer common substring between q and w (this is always fulfilled for $\delta = 1$), then we use Property 3.3.2 and otherwise Property 3.3.3. In practice this resulted in substantially improved running times compared to using only one signature. Now we consider the following problem. Given a word w, a fixed dictionary W, and a threshold δ, we would like to efficiently compute all words in W that share a substring with w of length larger than some minimum as well as the length of the common substring. We will refer to this problem as the *exact substring dictionary search*.

The exact substring dictionary search problem can be solved by using a known method based on a *suffix tree*. A suffix tree is a compacted trie that represents all suffixes of a text. We employ a compacted trie that represents all suffixes of the words in W, where each leaf node is equipped with a list of word-ids that contain the corresponding suffix. Each path in the tree following a suffix of q corresponds to a common substring between q and a subset of words in W. In general terms, the algorithm works by traversing the tree for each suffix of q. A disadvantage of this method is the size of the index. For example, the size of a suffix tree in practice is typically about 20 times the size of the text it represents.

Boytsov [2011] proposes a simpler and more space efficient method for exact matching of short substrings based on length-divided q-gram indexes. However, this method is not suitable for the longest common substring signature where the substrings can have arbitrary lengths. We employ a method that is a weak version of the suffix tree method from above based on a *permuted lexicon*. A permuted lexicon is a simple but practical data structure that supports efficient substring search in a dictionary of words [Bratley and Choueka, 1982]. It was used in Zobel and Dart [1995] as a heuristic to find likely approximate matches by investigating small neighborhoods of words in the permuted lexicon that correspond to substrings in q. We extend this technique to a full fuzzy matching algorithm by combining it with the longest common substring and pattern partitioning signatures.

3.3.2. Algorithm Based on Permuted Lexicon

In what follows, we describe the permuted lexicon data structure in finer detail and describe how to address the exact substring dictionary matching problem by using a precomputed *lcp* array.[5]

Definition 3.3.4 (Rotation)**.** *Consider a word w with index i in W. A rotation r of w is the pair (i, j), where j is the* shift *and denotes the j-th* cyclic permutation *of w.*

When referring to a rotation r, we will consider its string representation, keeping in mind that r is equipped with a “pointer“ to its original word in the dictionary.

Definition 3.3.5 (Permuted Lexicon)**.** *Let W be a dictionary of words. A permuted lexicon of W (or* $\mathrm{pl}(W)$*) is the lexicographically sorted list of the rotations of each $w \in W$.*

Let $r_j(q)$ be the j-th rotation of q, where $j \in \{1, \ldots, |q|\}$ if the longest common substring is used as a signature and $j \in \{p_1, \ldots, p_{\delta+1}\}$ if pattern partition is used as a signatures, where $p_1, \ldots, p_{\delta+1}$ are the set of

[5] *lcp* stands for longest common prefix.

position of the partitioning of q. Let r'_i be the i-th rotation in $\text{pl}(W)$ obtained by performing a binary search for $r_j(q)$. Then all words in W that share a non-empty substring with q starting at position j, will correspond to a certain neighborhood around r'_i in $\text{pl}(W)$. To compute the length of the common substrings efficiently, we use the following two observations. First, $lcp(r_j(q), r'_i) \geq lcp(r_j(q), r'_{i+k})$, where $lcp(r_j(q), r'_i)$ is the length of the longest common prefix between $r_j(q)$ and r'_i and $k \in \{0, 1, 2 \ldots\} \cup \{0, -1, -2, \ldots\}$. Therefore, the length of the common substring decreases as we go above or below position i in $\text{pl}(W)$. Second, the following holds

Observation 3.3.6. $lcp(r_j(q), r'_{i+k+1}) = \min\{lcp(r_j(q), r'_{i+k}), lcp(i + k)\}$

where $lcp(i) = lcp(r'_i, r'_{i+1})$ is an array of the lcps of the adjacent words in $\text{pl}(W)$ precomputed during the initialization of the algorithm. Therefore, we can compute the lcps between $r_j(q)$ and the next word in the neighborhood (starting from r'_i) in constant time. All words seen along the way are marked to avoid redundant matches and computations. The scanning stops when the current substring length is less than the minimum substring length (Property 3.3.2) or less than the length of the current matching piece from q (Property 3.3.3).

The candidate matches that survive the filtering are subject to additional suffix filtering. To apply these filtering efficiently, we make use of the following two observations. First, a rotation of a word contains the same multiset of characters as the original word. Second, since our algorithm computes the maximal matching substrings, the unigram frequency filter has been already computed on the prefixes of the rotations and hence it should be computed only on their suffixes. The algorithm is identical to the suffix filtering algorithm from Section 3.2.4.

Our implementation of the permuted lexicon uses $5 \cdot |w|$ bytes per word. The first 2 bytes are used to encode the shift and the lcp array, and the last 3 bytes are used to encode the word id. If the space usage is a concern, one can use a compressed version of the permuted lexicon from Ferragina and Venturini [2007]. This method is based on the Burrows-Wheeler transform of concatenated rotations. The lcp array could be compressed by gap and elias-gamma code.

The speed of this algorithm strongly depends on the length of the current matching substring, which in turn depends on δ and $|q|$. Therefore, this algorithm is efficient in the following cases:

- When δ is small relative to $|q|$. One example for this scenario is when δ is dynamic. Another example comes from the similarity join literature where the records have average length of more than 100 characters and thresholds that often varies from 1 to 3 errors (for example, see Xiao *et al.* [2008a]; Gravano *et al.* [2001]). If the threshold is 3, the common substring length must be at least 33 characters long. In a q-gram-based algorithm, this would hypothetically correspond to using $q = 33$. This is very restrictive since only few string would share q-grams that long.

- When the distance threshold is 1 as the common substring length must be at least $|q| - 1$ characters long. For example, this is as effective as the basic `DeleteMatch` algorithm from the previous section.[6]

This algorithm is less efficient when the longest common substring is less than 4 characters since the number of candidate matches becomes large, similarly as in q-gram-based algorithms. However, unlike q-gram-based algorithms, the words sharing longer substrings are found early and not considered again. In what follows, we include a number of additional optimizations to our algorithm.

3.3.3. Mismatching Rotations

Given two words w_1 and w_2, a substring s_1 in w_1 matches a substring s_2 in w_2 with respect to LD, iff s_1 is transformed into s_2 after performing the sequence of edit operations that transform w_1 into w_2. If two substrings match, then the difference in their positions must be at most δ [Sutinen and Tarhio, 1995]. The above algorithm finds all common substrings between q and words in W longer than certain length. This includes the substrings that do not match. We can easily skip such rotations as follows. Given a rotation r, let $sh(r)$ be the shift of r. Let r be the current query rotation and r' be the currently investigated rotation. If $|sh(r) - sh(r')| > \delta$ we can safely ignore r' since it is guaranteed that r and r' correspond to substrings in w_1 and w_2 that do not match.

[6] This algorithm, however, has a larger constant factor than *DeleteMatch*.

3.3.4. Clustered Errors

For another filter, consider a query q and a word w with $\mathrm{WLD}(q, w) = t \leq \delta$. If q and w have a substring of length $\max\{|q|, |w|\} - t$ in common, then they must share only a single substring. As a result, the errors must have adjacent positions, i.e., they are "clustered". However, the opposite is not always valid. Let r be the current query rotation, let r' be the currently investigated rotation of a word w and suppose $\max\{|r|, |r'|\} - lcp(r, r') = t \leq \delta$. The following example shows that we cannot immediately conclude that $\mathrm{WLD}(q, w) = t$.

Example 3.3.7. *Consider q=`algorithm` and w=`lgorithma` and consider the rotations in q and w with equal string representation `algorithm` for both words. Observe that* $\max\{|r|, |r'|\} - lcp(r, r') = |9 - 9| = 0$*, however,* $\mathrm{WLD}(q, w) = 2$.

Additional caveat exists when $\delta \geq 3$, in which case clustered errors cannot be detected merely based on the substring length as shown in the following example.

Example 3.3.8. *Let q=`algorithm` and w=`xyzgorithm` and let $\delta = 3$. Observe that q and w share the substring `gorithm` of length 7 and that $|7 - 10| = 3 \leq 3$. However, if `y=l` in w, then q and w would share additional substring, namely `l`. Hence, the errors would not be clustered anymore and* $\mathrm{WLD}(q, w) = 2$ *instead of the alleged value 3 (although* $\max\{|r|, |r'|\} - lcp(r, r')$ *remains 7).*

In the following, we define the notion of clustered errors more formally.

Definition 3.3.9. *Let w_1 and w_2 be two words with rotations r_1 and r_2 such that* $\max\{|r|, |r'|\} - lcp(r, r') \leq \delta$. *The errors in w_1 (respectively w_2) are clustered if w_1 can be partitioned into two substrings such that all errors are contained in only one of the substrings.*

Given a rotation r of w, let $r[i] = (sh(r) + i) \mod |r|$. Let w_1 and w_2 be two words with matching rotations r_1 and r_2. We distinguish among 3 cases of clustered errors in w_1 and w_2:

- The errors are in the beginning of w_1 and w_2 (e.g., `xxgorithm` and `ylygorithm`). This case takes place when the common substrings are suffixes of w_1 and w_2. It can be detected by verifying whether $r_1[lcp(r_1, r_2)] = lcp(r_1, r_2)$ and $r_2[lcp(r_1, r_2)] = lcp(r_1, r_2)$;
- The errors are in the end of w_1 and w_2 (e.g., `algoritxxx` and `algorithm`). Similarly, this case takes place when the common substrings are prefixes of w_1 and w_2. It can be detected by verifying whether $r_1[0] = 0$ and $r_2[0] = 0$;
- The errors are in the middle of w_1 and w_2 (e.g., `algoxithm` and `algoyyithm`). This case takes place when neither 1. nor 2. are satisfied and $w_1[0] = w_2[0]$.

Lemma 3.3.10. *Assume that the errors in q and w are clustered. Assume also that* $\max\{|r|, |r'|\} - lcp(r, r') \leq \delta$*, where r and r' are the matching rotations in q and w respectively. Then* $\mathrm{WLD}(q, w) \leq \delta$. *If, in addition,* $\max\{|r|, |r'|\} - lcp(r, r') = t < 3$*, then* $\mathrm{WLD}(q, w) = t$.

Proof. For the first part of the lemma, assume that q and w are partitioned as $q_1 \cdot q_2$ and $w_1 \cdot w_2$ such that $q_1 = w_1$. Since q can be transformed into w by transforming q_2 into w_2 and $|q_2| \leq \delta$ and $|w_2| \leq \delta$, it follows that $\mathrm{WLD}(q, w) \leq \delta$. A similar conclusion can be drawn for the other two partitionings of q and w. There are two possibilities for the second part of the lemma. If $\max\{|r|, |r'|\} - lcp(r, r') = 1$, then obviously $\mathrm{WLD}(q, w) = 1$ because q and w differ in only one character. Assume the two errors have positions $p_1 < p_2$ that are not consecutive, i.e., $1 + p_1 < p_2$. Then q and w must have a common substring with positions between p_1 and p_2. However, this would imply that $\max\{|r|, |r'|\} - lcp(r, r') > 2$. Hence, p_1 and p_2 must be consecutive positions. Consequently, $\mathrm{WLD}(q, w) = 2$. □

3.3.5. Early Stopping Heuristic

One way to address the problem of large number of candidate words when the longest common substring is short is to employ an early stopping of the scanning of the current neighborhood as follows. If the current substring length is short (e.g., less than 5 characters), then the scanning of the current neighborhood is terminated if no similar word is found after certain number of distance computations (cut-off parameter). The hope is that the potentially missed similar words matching the current (short) substring, will match either on a longer or on a less frequent subsequent substring.

Table 3.7.: Average number of words that must be scanned for different longest-common-substring signature lengths (Lemma 3.3.1) when the threshold is 2 on the dictionary of the DBLP dataset with 600K words.

Minimum substring length	Average number of words visited without a heuristic	Average number of words visited with a heuristic
2	240,023	15,771
3	44,233	8,019
4	18,596	5,376
5	10,052	3,669
6	5,590	2,818
7	3,689	2,261
8	1,529	1,529

Example 3.3.11. *Suppose $\delta = 2$ and q=`believe` and assume the algorithm is currently scanning the words that share a substring starts at position* 0 *in q. Say that the current substring length is 3 (the longest substring length is 3 as well) and the word w=`beleive` (with common substring `bel`) has been missed, although* $\mathrm{WLD}(q, w) \leq \delta$*. However, since q and w have the longer substring `vebel` in common as well, it is more likely that w will be found later.*

The above heuristic results in substantial decrease in the number of words visited (see Table 3.7). Moreover, the running times are almost one order of magnitude less for $\delta \geq 2$ when the longest common substring is short. This goes well with the fact that only a small fraction of the visited words matching a short substring (q-gram) have WLD within the threshold. The price paid for using this heuristic is a loss of recall. To achieve a recall of 95% a proper cut-off parameter must be set. Unfortunately, the value of the cut-off parameter strongly depends on the size of the dictionary as well as on its nature. In practice, we use an empirically precomputed values for a range dictionary sizes.

3.4. Experiments

We tested our algorithms on the word dictionaries of two of our test corpora, namely the DBLP corpus and the Wikipedia corpus (details on the test corpora are given in Chapter 5, Section 5.8). Both dictionaries were initially cleaned from obvious garbage. We used three parameters to identify garbage words: the number of digits in the word, the longest run of a single character and the longest run of non-vowel characters. As garbage we considered all words for which at least one of the following applies: (i) the number of digits is larger than one half of the total number of characters; (ii) the longest run of a single character is longer than 5; (iii) the longest run of non-vowel character is longer than 5. The size of DBLP's dictionary resulted in around 600K words and the size of Wikipedia's dictionary in around 9M words.

Our algorithms were implemented in C++ and compiled with GCC 4.1.2 with the -O6 flag. The experiments were performed on a machine with an Intel(R) Xeon(R) model X5560 @ 2.80GHz CPU with 1 MB cache and 30 GB of RAM using a RAID file system with sequential read/write rate of up to 500 MiB/s. The operating system was Ubuntu 10.04.2 in 64-bit mode.

3.4.1. Algorithms Compared

We compared the following algorithms:

1. `DeleteMatch` (Section 3.2), an algorithm based on truncated deletion neighborhoods combined with unigram frequency filtering. The truncation length parameter (k) was set to 7 characters. The worst-case running time of this algorithm (without taking into consideration the distance computations) is $\binom{k}{\delta} \cdot N(k - \delta)$, where $N(i)$ is the maximum number of words in W containing any i characters.

2. `PermuteScan` (Section 3.3), a sequence-filtering algorithm that combines the longest common substring signature with the pattern partitioning signature with index based on permuted lexicons. The worst-case

Table 3.8.: Index size given as a percentage of the dictionary size (the sizes of the dictionaries of the DBLP and the Wikipedia corpus were 4 MB and 84 MB respectively).

	DBLP	Wikipedia
DeleteMatch	575% ($k = 7$) or 200% ($k = 6$)	267% ($k = 7$) or 93% ($k = 6$)
PermuteScan	500%*	500%*
DivideSkip	450%	350%
msFilter	687%	550%

running time of this algorithm is $O(\min\{n \cdot N(\lceil n/\delta \rceil - 1), n/\delta \cdot N(\lfloor n/(\delta + 1) \rfloor)\})$, where n is the length of the pattern (query). We used the approximate version in our experiments with a recall of 95%.

3. `DivideSkip` (Section 3.1), a state-of-the-art algorithm based on merging q-gram lists [Li *et al.*, 2008] employing skipping techniques to improve the performance of the standard algorithm. The q-gram length was set to 3 characters for $\delta = 1$ and to 2 characters for $\delta = 2$. q-gram based algorithms that employ skipping optimizations to improve the running time have not been extensively compared with other methods in the literature. In the original work this algorithm has competitive running times and outperforms the standard q-gram based algorithms. The goal was to compare this algorithm with our own sequence-filtering algorithm. The running time of this algorithm in the worst-case is $O(n \cdot N(q))$, where q is the q-gram length.

4. `msFilter` (Section 3.1), a trie-based algorithm that uses an additional trie built over the reversed strings to launch a series of $\delta + 1$ subqueries. It combines traversing tries with pattern partitioning and Leveneshtein automata to control the trie traversal [Mihov and Schulz, 2004]. In addition, it employs deletion neighborhoods based on minimal transducers to improve the performance for $\delta = 1$. In the extensive survey of Boytsov [2011], this algorithm (called FB-trie) is the most efficient of all algorithms with a practical index.[7] Its worst-case running time is equivallent to the worst-case running time of full neighborhood generation algorithms, i.e., $O(\delta \cdot n^{\delta/2+1}|\Sigma|^{\delta/2})$.

5. `PrefixTrie` (Section 3.1) is an algorithm based on traversing a prefix trie, originally designed for incremental fuzzy word / prefix matching [Ji *et al.*, 2009]. We include it here for completeness. More details (including space usage) is given in Section 4.5.

3.4.2. Queries

To ensure that our query generation method does not significantly influence the performance of the algorithms, we evaluated the efficiency of our algorithms by generating queries in two different ways.

With our first method we generated 1000 random distinct single-word queries per test collection. Each word was picked uniformly at random without repetition, i.e., with probability proportional to its term frequency (stop-words were excluded). We applied a random number of 0 to 3 errors to the sampled words as follows. To each word we applied a single error (at a random position) with probability p. To each picked word longer than 5 characters, we applied an additional error with the same probability. The same procedure was repeated for words longer than 10 characters. Each type of error (deletion, insertion, substitution) was applied with equal probability. We set $p = 0.3$ in our experiments (note that in general, smaller p results in larger running times for all algorithms).

With our second method we generated misspellings from the dictionary of the corresponding collection. We first picked a frequent word uniformly at random from the dictionary (without repetition), found all of its misspellings and picked a word uniformly at random from this set of words (including the frequent word).

For each query we measured the time to find all similar words in the dictionary of the corresponding collection by using different distance thresholds. According to our experiments, the running times of the algorithms were not sensitive to the query generation method but rather on the total number of matches. All of the algorithms in general required larger running times on query sets that contained larger proportion of valid words.

[7] We tested an implementation provided by the authors of Mihov and Schulz [2004].

Table 3.9.: Average fuzzy word matching running times using (fixed) thresholds of 1 and 2 errors as well as dynamic threshold that depends on the length of the query and varies from 1 to 3 errors (1 for $|q| \leq 5$, 2 for $6 \leq |q| \leq 10$, and 3 otherwise).

	DBLP			
	$\delta = 1$	$\delta = 2$	$\delta = 3$	$\delta = 1 - 3$
DeleteMatch	**0.01** ms	**0.15** ms	**1.6** ms	**0.5** ms
PermuteScan	0.07 ms	6.5 ms	23.0 ms	1.4 ms
DivideSkip	0.4 ms	5.4 ms	-	4.4 ms
msFilter	0.07 ms	0.8 ms	11.0 ms	-
PrefixTrie	0.4 ms	19.1 ms	-	58.3 ms
	Wikipedia			
	$\delta = 1$	$\delta = 2$	$\delta = 3$	$\delta = 1 - 3$
DeleteMatch	**0.04** ms	**1.5** ms	**20.4** ms	**4.0** ms
PermuteScan	0.3 ms	52.0 ms	424.5 ms	14.1 ms
DivideSkip	11.4 ms	87.0 ms	-	65.3 ms
msFilter	0.4 ms	3.4 ms	70.6 ms	-
PrefixTrie	1.0 ms	54.8 ms	-	132.4 ms

3.4.3. Discussion

The results are shown in Table 3.9.[8] `DeleteMatch` was the fastest algorithm in general, with `PermuteScan` being a competitive option when δ is dynamic or when $\delta = 1$ and `msFilter` being a competitive option when $\delta = 2$. `PermuteScan` had a small advantage over `msFilter` for $\delta = 1$, but it was outperformed by a large margin for $\delta = 2$.

The bottleneck of `PermuteScan` lies in the $N(i)$ factor of its running time. More specifically, when $i \leq 3$ the algorithm must visit all words than contain a substring of length at least 3, which, first, is very large in practice, and second, depends linearly on the dictionary size. The same holds true for `DevideSkip` since its skipping technique improves the running time only for a small factor. In fact, any algorithm that is based on q-grams or pattern partitioning that indexes short substrings would suffer from the same problem. The large advantage of `PermuteScan` over `DivideSkip` diminishes for $\delta \geq 2$. `PermuteScan` did not perform well for $\delta = 3$ since, again, its effectiveness depends heavily on the ratio between the average word length and δ. We note that it would be interesting to see how `PermuteScan` performs on strings with lengths much larger compared to δ. Figures for `DivideSkip` for $\delta = 3$ were omitted since the complete result set was not computed by this algorithm. The $N(i)$ factor in the running time of `DeleteMatch` is low enough for the algorithm to be efficient as i is kept larger than 3 in all settings.

Since `msFilter` is based on trie traversal, its running time is sensitive to the alphabet size. This is because these algorithms typically must also visit all children of a given node that is being visited. Our experiments were carried out on an alphabet that contains only the 26 english letters `a...z`. On a larger alphabet (e.g. unicode) we would expect the performance of this algorithm to degrade substantially. Another setting where `msFilter` would be slow are dictionaries that are prefix dense, i.e., contain many short words. This is because the prefix representation in that case is less effective, each node has a lot of children and hence the algorithm must visit a lot of nodes near the root of the trie. Yet another unfavorable setting is when the pattern is short because the algorithm is based on launching subqueries using only one half of the pattern. An advantage of `msFilter` is that it scales very well on large dictionaries.

Tale 3.8 shows the index size of each algorithm given as a percentage of the dictionary size. Somewhat surprisingly, `DeleteMatch` was the most space efficient algorithm. Its index size for $\delta \leq 3$ with $k = 6$ was below the size of the dictionary at the price of less than twice higher average running times compared to choosing $k = 7$. For `PermuteScan` we used an implementation that does not employ compression. A compression method based on Burrows-Wheelers transform from Ferragina and Venturini [2007] in practice achieves space occupancy well below the dictionary size. `msFilter` had the largest index size in practice. We note that a variant of this algorithm implemented in Boytsov [2011] (without deletion neighborhoods for $\delta = 1$) required around 300% of the vocabulary size.

[8] The results on the GOV2 collection were similar to those on Wikipedia and are omitted.

4. Efficient Fuzzy Prefix Matching

In this chapter, we focus on the fuzzy prefix matching problem. The fuzzy prefix matching problem is similar but computationally harder than the fuzzy word matching problem, which was the subject of the previous chapter. The reason lies in the typically much larger result size of an instance of the fuzzy prefix matching problem. For example, the number of fuzzy completions for the 4-letter prefix `algo` on the English Wikipedia with threshold set to 1 is around 14K, while the number of words similar to the word `algorithm` with threshold set to 2 is around 100.

We start the chapter by summarizing the related work in Section 4.1. In Sections 4.2 and 4.3, we present two fast and practical off-line algorithms based on the algorithms presented in the previous chapter. In Section 4.4, we propose a simple incremental algorithm and argue that it is sufficiently fast in practice when the prefix is longer than 5 characters. In Section 4.5, we present our experimental results.

4.1. Related Work

Unlike fuzzy word matching, fuzzy prefix matching arises as a relatively new problem in the literature. Two similar approaches, both based on a trie data structure, have been independently proposed in Chaudhuri and Kaushik [2009] and Ji *et al.* [2009]. The algorithm from Ji *et al.* [2009] maintains a set of *active nodes* that represent the set of nodes corresponding to prefixes within the distance threshold. The set of all leaf descendants of the active nodes are the answer to the query. At the beginning all nodes with depth less or equal than the threshold are set as active. The algorithm is incremental, which means to compute the active nodes for the prefix $p_1 \dots p_n$ for $n > 0$, we first have to compute the active nodes for the prefix $p_1 \dots p_{n-1}$. The set of new active nodes is computed from the set of old active nodes by inspecting each child node and differentiating between two cases: substitution (when p_n is different than the corresponding character of the child node) and a match (when p_n is equal to the corresponding character of the child node).

A weakness of this algorithm is the large number of active nodes that have to be visited. For example, for the initialization of the algorithm alone, we must visit all nodes with depth less or equal than the threshold. Furthermore, computing the answer set incrementally is only fast when the user types the query letter by letter. If a relatively long query (e.g., 7 letters) is given, the algorithm must compute the answer set for each prefix first. This means that computing the answer set of a long query is more expensive than computing the answer set of a short query, although long queries have much smaller answer sets. Another disadvantage is that a traversal of all subtrees corresponding to active nodes is expensive and requires a significant fraction of the total time (up to one half according to Ji *et al.* [2009]).

4.2. Algorithm: DeleteMatchPrefix

In this section, we first show how the fuzzy word matching algorithm from Section 3.2 can be naturally extended to a fuzzy prefix matching algorithm (Section 4.2.1). We then propose a few optimization techniques that will significantly improve the running time of our algorithm (Sections 4.2.2 and 4.2.3).

4.2.1. The Basic Algorithm

Recall that Lemma 3.2.3 from Section 3.2 allowed us to use the subsequences obtained from the δ-deletion neighborhood of a word as signatures to find the candidate matching words with WLD within δ. Moreover, Lemma 3.2.4 and Lemma 3.2.5 allowed us to conceptually truncate each word to its k-prefix and index only the δ-subsequences of the prefixes instead of the full deletion neighborhood of each word. It is not hard to see that as a by-product, besides the result set, our algorithm will find all words that contain prefixes with WLD at

Table 4.1.: Average number of distance computations per match for different query word lengths when $\delta = 1$ by using the *DeleteMatchPrefix* algorithm with different filters (on the dictionary of the DBLP collection with around 600K distinct words). First row: no filters, second row: using the optimization from Lemma 4.2.3, third tow: using all filters.

4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
1.5	1.4	1.2	1.2	2.1	3.2	4.4	5.6	7.0	9.9	13.5
0.5	0.5	0.4	0.4	1.4	2.4	3.6	4.9	6.3	9.2	12.6
0.5	0.5	0.4	0.4	0.7	0.8	0.9	0.9	1.0	1.0	1.0

most δ from q, given that $|q| \geq k$. According to Definition 2.1.2, these are the words from W with PLD at most δ from q, given that $|q| \geq k$.

Lemma 4.2.1. *Let q be a prefix and w a word with* $\mathrm{PLD}(q, w) \leq \delta$ *and let $q[k]$ and $w[k]$ be their k-prefixes such that $k \leq |q|$. Then $\exists s \in U_d(q[k], \delta) \cap U_d(w[k], \delta)$ with $|s| \geq k - \delta$.*

Proof. According to the definition of PLD (Definition 2.1.2), there is a prefix $w' \preceq w$ such that $\mathrm{WLD}(q, w') \leq \delta$. If we ignore for a moment the suffix of w with length $|w| - |w_p|$ and consider q and w' as words, according to Lemma 3.2.4, $\exists s \in U_d(q[k], \delta) \cap U_d(w'[k], \delta)$ with $|s| \geq k - \delta$. □

Example 4.2.2. *Consider a prefix q=*`algoxx` *and let $\delta = 2$. Consider the word* `algorithm`*. If $k = 6$, then the 6-prefix* `algori` *will match with q on the subsequence* `algo`*. However, if $k = 7$, then there is no common subsequence between* `algoxx` *and the 7-prefix* `algorit` *in their 2-deletion neighborhoods.*

The latter gives rise to the following algorithm. In addition to the k-prefixes, index all i-prefixes for $i = m, \ldots, k$, where m is the minimum prefix length that we would like to consider. At query time proceed as in Section 3.2, with the difference that PLD instead of WLD is used to verify the candidate matches. Observe that Lemma 3.2.5 ensures no overlap of signatures of the same word for different prefix lengths. This is because for each i, only the subsequences of length $i - \delta$ are generated. For example, suppose $\delta = 2$ and $k = 7$. We would then have subsequences of length 5 for $i = 7$, subsequences of length 4 for $i = 6$, etc. To reduce the space of our algorithm we make use of the observation that the correspondig subsequences in the deletion neighborhoods of the i-prefixes are also prefixes of each other. For example, consider the prefixes of the word `algorithm` starting from $i = 7$: `algorit`, `algori`, `algor`, etc. Now consider all subsequences obtained by deleting the second character: `agorit`, `agori`, `agor`, etc. Since they are prefixes of each other, they can be efficiently represented by using a prefix trie. Each of these strings would then be represented by a single path in the trie following `agorit`. To represent the inverted lists efficiently, we store only the inverted lists corresponding to leaf nodes in the trie (subsequences of a prefix of length k). To access the inverted list of an internal node (subsequence of a prefix of length less than k), we simply traverse the subtrie rooted at that node.

4.2.2. Prefix Filtering

Given a prefix q and two words w and w', suppose it has been already determined that $\mathrm{PLD}(q, w) = t \leq \delta$. It is intuitive that if w and w' share a prefix that is long enough, then $\mathrm{PLD}(q, w')$ should be t as well. The following lemma gives us the simple but efficient means to skip the verification of candidates that share long prefixes by using only the precomputed list of lcps between adjacent words in W.

Lemma 4.2.3. *Consider two words w_1 and w_2 and assume that $w_1 > w_2$. Let* $\mathrm{PLD}(q, w_1) = t \leq \delta$*, let $lcp(w_1, w_2)$ be the length of the longest common prefix between w_1 and w_2 and let l be the last position in w_1 such that* $\mathrm{PLD}(q, w_1[l]) = t$*. If $l \leq lcp(w_1, w_2)$, then* $\mathrm{PLD}(q, w_2) = t$*.*

Proof. Assume that $l \leq lcp(w_1, w_2)$. It is obvious that $\mathrm{PLD}(q, w_2)$ cannot be larger than t. It is also not hard to see that if $l < lcp(w_1, w_2)$ and $\mathrm{PLD}(q, w_1) = t$ then $\mathrm{PLD}(q, w_2) = t$. Let $l = lcp(w_1, w_2)$. Assume that $l < |w_1|$. Recall that the dynamic programming table has the property that the minimum value in any column is non-decreasing (this follows directly from the recursion of the Levenshtein distance). Let dp_{w_1} be the dynamic programming table for computing the Levenshtein distance between q and w_1 and recall that $\mathrm{PLD}(q, w_1) = \min\{dp_{w_1}[|q|, i]\}, 1 \leq i \leq |w_1|$. Observe that $dp_{w_1}[|q|, l]$ has the minimum value in the l-th column since $dp_{w_1}[i, l]$, $i < |q|$ corresponds to the Levenshtein distance between a prefix of q and $w_1[l]$, which cannot decrease. Hence, $dp_{w_2}[|q|, i]$, $i > l$ cannot decrease. Consequently, $\text{PLD}(q, w_2) = t$. If $l = |w_1|$, then $\text{PLD}(q, w_2) < t$ would require $w_1 \leq w_2$, however, this is not possible since by assumption $w_1 > w_2$. □

Remark 4.2.4. *If the exact value of* WLD(q, w) *is not required, then l above could be chosen as the first position in w_1 such that* $\text{PLD}(q, w_1[i]) = t$ *instead of the last. The above optimization in this case is more effective since it is more likely that the condition $l \leq lcp(w_1, w_2)$ would be satisfied.*

Example 4.2.5. *Suppose q=*`tren` *and consider the words* `transport`*,* `transition`*,* `transformation` *and* `transaction` *(given in lexicographically decreasing order). Observe that the* PLD *between* `tren` *and* `transport` *is 1 and that the conditions from Lemma 4.2.3 are satisfied on the next three words. This means that without performing distance computations, we can safely conclude that the* PLD *between q and each of these words is 1 as well.*

In general, if we have already determined that $\text{PLD}(q, w_i) = t \leq \delta$ for some word w_i by using a distance computation, then we can skip the verification on the words adjacent to w_i in the current inverted list as long as the conditions in Lemma 4.2.3 are satisfied. Note that W should be given in lexicographically decreasing order and we should be able to compute the lcps between a word w_i and any adjacent word w_{i+k} for $k > 0$ in constant time. This can be done similarly as in Section 3.3 by computing a lcp array and then using Property 3.3.6.

4.2.3. Suffix Filtering

As before, word truncation involves generating more false-positive candidate matches on words longer than k characters since the suffixes of the candidate words are ignored. Consider a prefix q and a candidate word w such that $|w| > k$, sharing a subsequence of length $k - \delta$. As in Section 3.2.4, by using Lemmas 3.2.6 and 3.2.7, we compute the unigram frequency distance between q and w starting at position $l + 1$ and ending at positions $|q|$ in q and $\min\{|w|, |q| + \delta\}$ in w.

Table 4.1 shows the average number of distance computations per computed match with and without using the above optimizations. It can be seen that their effect is complementary: the optimization from Lemma 4.2.3 is more effective on short queries due to the long common prefix relative to q, while the character overlap filter is more effective on longer queries due to the effect on truncation.

4.3. Algorithm: PermuteScanPrefix

In this section, we introduce a corresponding longest common substring signature for the prefix Levenshtein distance (Section 4.3.1). As before, we formulate the problem as an exact substring matching problem and present an algorithm for its solution (Section 4.3.2). We then propose optimizations to reduce the space usage of the algorithm (Section 4.3.3).

4.3.1. The Longest Common Substring Signature on Prefixes

The key property from Section 3.3 (given in Lemma 3.3.1) allowed us to use the length of the longest substring between two strings as a filter to generate candidate matches. However, the following example shows that this property cannot be used directly if q is a prefix.

Example 4.3.1. *Consider q=*`algor` *and w=*`alxgorxthmic`*. Since* $\text{PLD}(q, w) = 1$*, according to Lemma 3.3.1, q should share a substring of length at least 5 with a prefix of w. Note that by using a permuted lexicon from Section 3.3, we can only find the common substrings* `al` *and* `gor` *between q and w, but not the substring* `algor` *shared with the prefix* `alxgor` *of w.*

The following related lemma is an equivalent of Lemma 3.3.1 on prefixes.

Lemma 4.3.2. *If* $\text{PLD}(q, w) \leq \delta$*, then there exist a prefix p of w, such that p and q share a substring of length at least* $\lceil |q|/\delta \rceil - 1$.

Proof. The proof is a direct consequence of Lemma 3.3.1 and Definition 2.1.2 (prefix Levenshtein distance). □

Analogously to Section 3.3, given a prefix q, a dictionary of words W and a threshold δ, we would like to efficiently find all words in W whose prefix shares a long enough substring with q.

4.3.2. Index Based on Extended Permuted Lexicon

In the following, we introduce the notions of *rotation* and *permuted lexicon* that are analogous to those from Section 3.3.2.

Definition 4.3.3 (Rotation). *A rotation r is a triple (i, j, k) denoting the k-th cyclic permutation of the j-th prefix of the i-th word in W.*

Definition 4.3.4 (Extended Permuted Lexicon). *Let W be a dictionary of words. An extended permuted lexicon of a word dictionary W consists of all rotations of the prefixes of the words in W, sorted in lexicographic order.*

The basic algorithm is similar to that from Section 3.3.2 and it goes in three main steps:

1. Given a dictionary of words W and a prefix q, perform a binary search for $r_i(q)$ in the extended permuted lexicon of W to find the neighborhood of words with prefixes that share a substring with q starting at position i;

2. Scan the neighborhood and verify the candidate matches (use Property 3.3.6 to compute the substring lengths in constant time and use Lemma 4.2.3 to skip the verification on adjacent candidate matches that share long prefixes);

3. Stop the scanning when the condition from Lemma 4.3.2 is not fulfilled.

4.3.3. Compacting the Extended Permuted Lexicon

If constructed in the straightforward way, the extended permuted lexicon will contain rotations with identical string representations multiple times. For example, the words `algorithm`, `algorithms` and `algorithmic` will generate three rotations with identical string representations for each prefix of `algorithm`. We call such rotations *equivalent*. More specifically, as equivalent we will define all rotations with equal shift and equal string representation.

Assume that W is given in lexicographically sorted order. Each set of words in W that shares a common prefix of length n will generate identical rotations of length n. We store each such set of words only once by observing that for every identical rotation we need to store only the index of the first word and the number of adjacent words that share the corresponding prefix of length n. To achieve this, we initially construct a trie over W such that each node in the trie is augmented with the list of word-ids that share the corresponding prefix. Then we traverse the trie, and for each unique prefix p we generate all of its rotations.

Recall that word truncation was already used in Section 3.2 to reduce the size of the index of signatures. We show that the same technique can be used to reduce the size of the extended permuted lexicon further.

Lemma 4.3.5. *Let* $\mathrm{PLD}(q, w) \leq \delta$ *and let* $q[k]$ *and* $w[k]$ *be the k-prefixes of q and w respectively, for some $k > \delta$. Then $w[k]$ and $q[k]$ share a substring of length at least* $\lceil |q[k]|/\delta \rceil - 1$.

Proof. Relevant for us are only the truncated characters in q and w that affect common substrings. The only effect truncating a single character has is shrinking the affected substrings between q and w. This cannot introduce new errors in the strings that could potentially generate new (shorter) substrings between q and w. Hence, $w[k]$ and $q[k]$ must share a substring of length at least $\lceil |q[k]|/\delta \rceil - 1$. □

As before, word truncation will negatively influence the running time of the algorithm due to the reduced signature length.

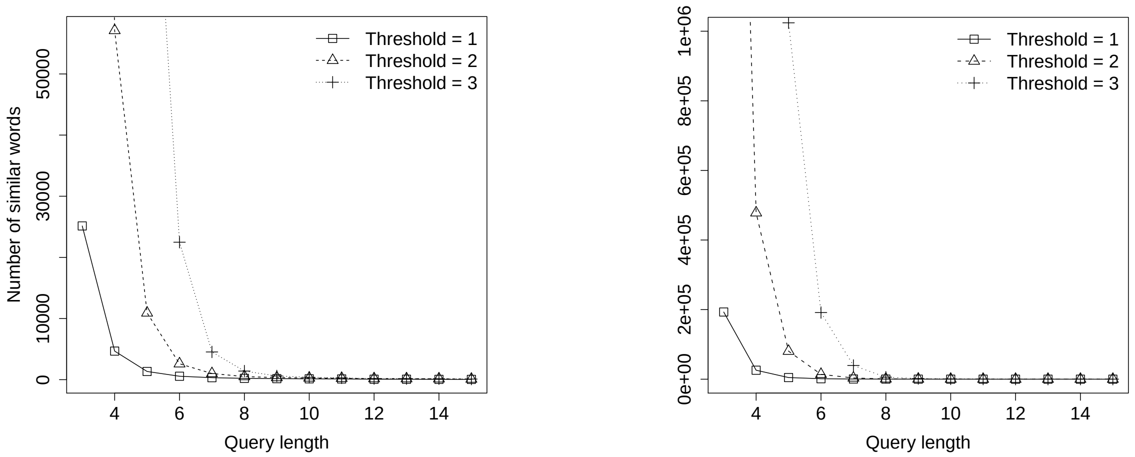


Figure 4.1.: Average number of matches for different prefix lengths on the dictionaries of the DBLP collection with 600K words (left) and the Wikipedia collection with 28.5M words (right) when using prefix Levenshtein distance with thresholds 1, 2 and 3.

4.4. Incremental Fuzzy Prefix Matching

Incremental algorithms are effective when the user types the query letter by letter and when a sufficiently large prefix of the to-be-completed query word has already been typed in. This is, first, because the candidate matching words are already drastically reduced, and second, because computing the result set from previous results is computationally much less expensive than computing the result set from scratch. Computing the result set incrementally is more expensive when the user has already a longer part of the query in mind. This is because the algorithm has to compute the result set for each prefix of the typed query. Furhermore, incremental algorithms are not too useful when the underlying corpus is large and the query is too unspecific (e.g., shorter than 4 letters) since the result set would be, on the one hand, too large to be useful, and on the other, too expensive to compute. For example, if W is very large, it is certainly too expensive to compute all similar completions to a prefix of length 1.

In what follows, we describe a simple yet practical incremental algorithm that can be sufficiently fast in practice and even competitive to the more sophisticated state-of-the-art algorithm from Ji *et al.* [2009] when the prefix is longer than 5 characters.

The main idea of the algorithm is as follows. Consider a query $q_i = p_1 \ldots p_i$ and a word w. Consider also the dynamic programming table used to compute the prefix Levenshtein distance between q and w and recall that only the cells that are at most δ cells away from the main diagonal are relevant for the computation (see Table 2.1). Say $\text{PLD}(q_i, w)$ has been already computed and the dynamic programming table has been already filled. Observe that to compute $\text{PLD}(q_{i+1}, w)$ where $q_i \leq q_{i+1}$, we only require the $2 \cdot \delta + 1$ non-empty cells from the last row of the dynamic programming table for w. Given a fixed dictionary W, a threshold δ and a query q_i, assume that all $w \in W$ with $\text{PLD}(p_i, w) \leq \delta$ have already been computed, and that an array $arr(w)$ of size $2 \cdot \delta + 1$ has been assigned to each w, where $arr(w)$ contains the last row of the dynamic programming table for w. Let $W_i \subset W$ be the result set for q_i. To compute W_{i+1}, for each $w_j \in W_i$ we update $arr(w_j)$ and compute $\text{PLD}(q_{i+1}, w_j)$ by using p_{i+1} and the values already stored in $arr(w_j)$. Observe that if $lcp(w_j, w_{j+1}) \leq |q_i| + \delta$, then $arr(w_{j+1}) = arr(w_j)$ and therefore no computation is required for w_{j+1}. The latter requires W in lexicographic order and computation of the lcp values in constant time, similarly as in Section 4.2.

The running time of the algorithm is $O(|W_i| \cdot \delta)$, where $|W_i|$ is the result size for q_i. We have observed empirically that $|W_i|$ becomes very small compared to $|W|$ when $|q| \geq 3 + \delta$ and practically constant for $|q| > 7$ (see Figure 4.1).

Table 4.2.: Index size given as a percentage of the dictionary size (the sizes of the dictionaries of the DBLP and the Wikipedia corpus were 4 MB and 84 MB respectively).

	DBLP	Wikipedia
DeleteMatchPrefix	375%	152%
PermuteScanPrefix	575%*	251%*
PrefixTrie	825%*	728%*

4.5. Experiments

In this section, we evaluate our fuzzy prefix matching algorithms. The algorithms were tested on the same word dictionaries from Section 3.4 (“clean“ versions of the word dictionary of the DBLP corpus with around 600K words and the word dictionary of the Wikipedia corpus with around 9M words). The experiments were carried out in setting identical to that from Section 3.4.

4.5.1. Algorithms Compared

We implemented and compared the following algorithms:

1. `DeleteMatchPrefix` (Section 4.2), an algorithm based on the longest common subsequence signature and a combination of filters. The truncation length parameter (k) was set to 6 characters;

2. `PermuteScanPrefix` (Section 4.3), an algorithm based on the longest common substring signature and a combination of filters. The truncation length parameter (k) was set to 8 characters;

3. `PrefixTrie` (Section 4.1), the incremental algorithm from Ji *et al.* [2009] based on a prefix-trie.

4.5.2. Queries

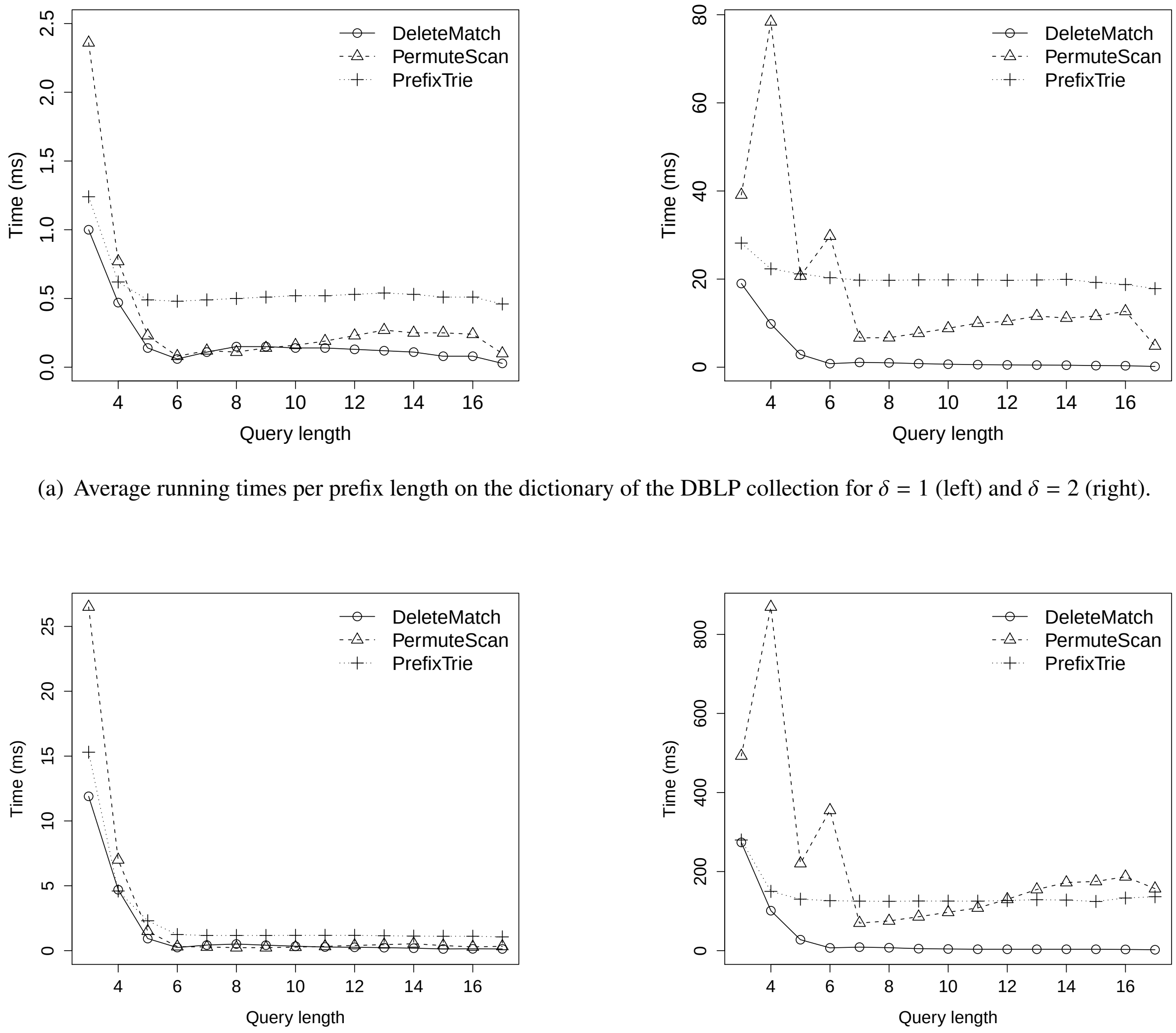
To evaluate the efficiency of our algorithms, we generated 1000 keywords per test collection as in Section 3.4.2, and from each keyword produced all prefixes starting from length 3. For each query, we measured the running time to compute and report all similar completions in the dictionary of the corresponding collection by using both, incremental and non-incremental fuzzy prefix matching.

4.5.3. Discussion

Figure 4.2 shows the average running times for different prefix lengths and different thresholds on two of our test collections. In summary, `DeleteMatchPrefix` was the fastest algorithm, with `PermuteScanPrefix` occasionally being a slightly faster option when $\delta = 1$. The running time of all algorithms was substantially larger on short prefixes due to the large number of matches. For example, when $\delta = 2$, the prefix `alg` matches all words that contain a prefix with at least one equal character at the same position. An incremental algorithm must compute these matches regardless of the length of the query. This makes `PrefixTrie` competitive only on short prefixes. `PermuteScanPrefix`, on the other hand, is competitive for $\delta = 1$ or when the prefix is at least 7 characters long. The slight increase in the running time of our algorithms around prefix lengths 7 and 9 was due to the effect of word truncation.

Figure 4.3 shows the average running times when the query is typed letter by letter starting from prefix length 3. In this experiment we used the simple incremental algorithm from Section 4.4 on prefixes longer than 5 characters (and non-incremental algorithm otherwise) and compared against `PrefixTrie`. The main observation is that for short prefixes `PrefixTrie` was hard to beat by a non-incremental algorithm. Hence, this algorithm remains the fastest option for incremental prefix matching. However, the difference diminishes as the query becomes longer than 5 characters where even a simple incremental algorithm can do well in practice.

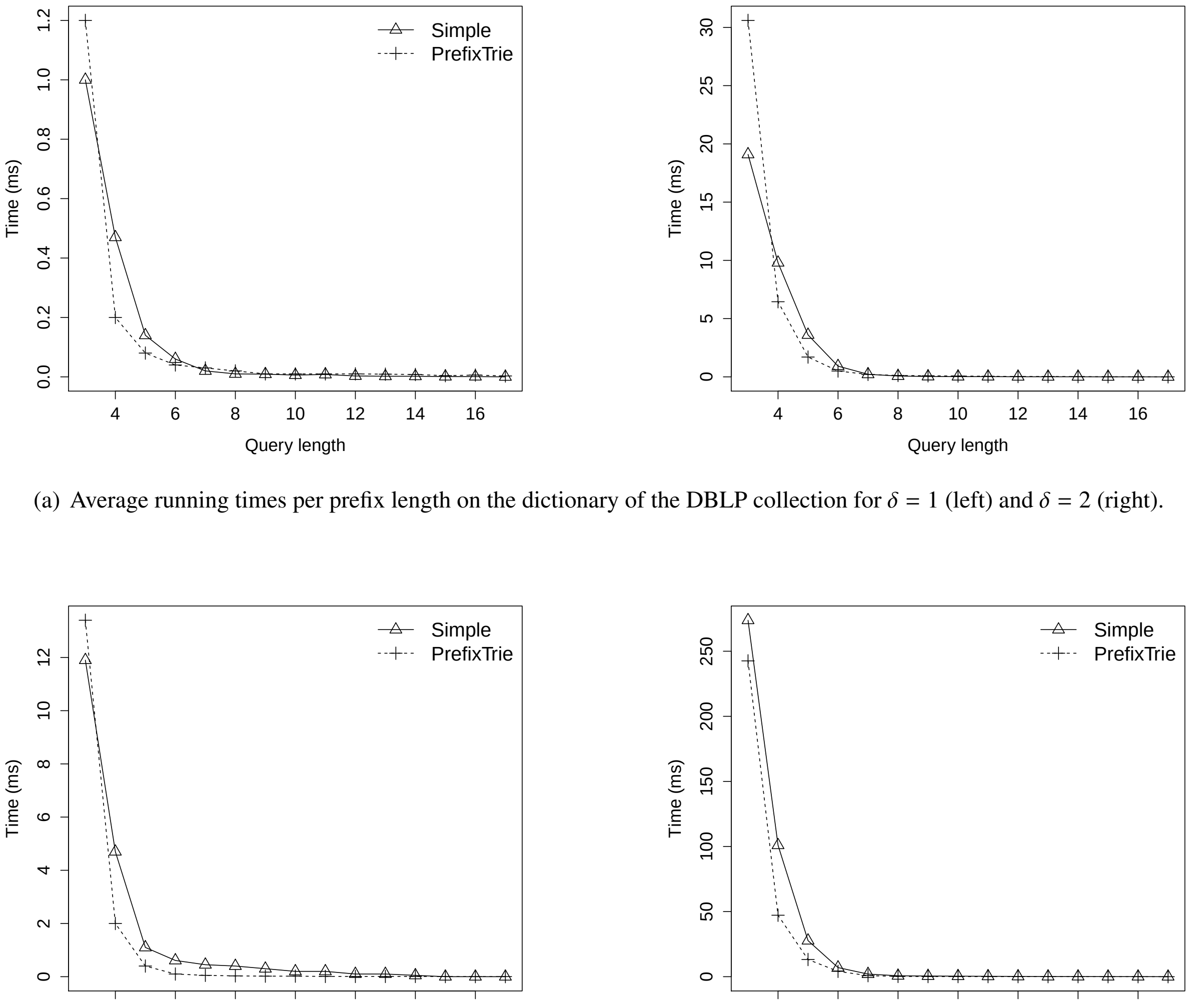
Figure 4.2 shows the index size of each compared algorithm given as a percentage of the dictionary size. Our implementations of `PermuteScanPrefix` and `PrefixTrie` did not employ compression.



(a) Average running times per prefix length on the dictionary of the DBLP collection for $\delta = 1$ (left) and $\delta = 2$ (right).

(b) Average runnings time per prefix length on the dictionary of the Wikipedia collection for $\delta = 1$ (left) and $\delta = 2$ (right).

Figure 4.2.: Average fuzzy prefix matching running times.



(a) Average running times per prefix length on the dictionary of the DBLP collection for $\delta = 1$ (left) and $\delta = 2$ (right).

(b) Average running time per prefix length on the dictionary of the Wikipedia collection for $\delta = 1$ (left) and $\delta = 2$ (right).

Figure 4.3.: Average fuzzy prefix matching running times by using incremental matching (the keyword is being typed letter-by-letter starting from length 3).

5. Efficient Fuzzy Search

In the previous two chapters, we have seen how to efficiently find all words similar to a given query word in a given dictionary. In this chapter we will describe how to use these results to do an actual fuzzy search.

In section 5.1, we start by familiarizing the reader with the related work. Then in Section 5.2, we give an overview of our algorithm and explain how we address the deficiencies of the baseline algorithm introduced in Chapter 2. In the following Sections 5.3 and 5.4, we introduce two novel index data structures called *fuzzy word index* and *fuzzy prefix index*. Then in Sections 5.5 and 5.6, we show how to efficiently solve the fuzzy search problem based on our new indexes. In Section 5.7, we show how to compute the result set incrementally from a previous result by using caching in time negligible compared to computing the result set from scratch. In Section 5.7, we provide the experimental results for our fuzzy search.

5.1. Related Work

5.1.1. Fuzzy Search

There is little work done on efficient fuzzy keyword-based search, let alone fuzzy prefix search. Ji *et al.* [2009] proposes an alternative solution to the intersection of union list problem given in Equation 2.1 (Section 2.3.1) which is an alternative formulation of the fuzzy search problem. Recall that the intersection of union list problem is to compute the intersection of two or more unions of lists. The basic idea in Ji *et al.* [2009] is to compute the intersection via so called *forward lists*. A forward list is the lexicographically sorted list of all distinct word ids in a document. A union list that corresponds to an exact prefix is the union of the inverted lists of all words that are completions of the prefix. Each union list consists of doc ids with a forward list stored for each doc id. Say we want to intersect two union lists that correspond to two exact prefixes. Provided that the word ids are assigned in lexicographic order, each prefix can be regarded as a range of word ids. The result list is computed as follows. First, we determine the shorter of the two union lists. Second, we determine the word ids (from the forward lists in the shorter union lists) contained in the word range corresponding to the longer union lists by performing a binary search in each forward list of the shorter union lists. A successful binary search means that a word within the word range corresponding to the longer list is contained in a document from the shorter list, i.e, the document is in the intersection. Hence, this procedure will result in the intersection of the two union lists.

If fuzzy prefix search is used, however, the union lists do not correspond to single word ranges anymore and the same procedure must be applied to each prefix within the distance threshold. The total number of such prefixes is usually large, rendering the algorithm expensive for fuzzy search (more details are given in Section 5.8.2).

5.1.2. Approximate String Matching

Approximate string matching (ASM) is a well studied and related, yet a distinctly different problem. The ASM problem is to find all occurrences O of a given pattern P in a long text T with $\mathrm{WLD}(O, P) \leq \delta$. Typical applications include indexing of DNA or protein sequences, signal processing and pattern matching, for example, retrieving encoded musical passages similar to a sample. The main differences to the fuzzy search problem are that there is no logical division of the text into documents and that there is no notion of word boundaries. Furthermore, the task in ASM is to simply report the approximate occurrences of a single pattern, while fuzzy search is to find the (ranked) set of documents that contain the query words approximately or exactly.

Indexes for approximate string matching usually come in three flavors. Those oriented to worst-case performance are based on backtracking a suffix tree or suffix array, taking into advantage the factoring of similar substrings achieved by suffix trees or arrays [Ukkonen, 1993; Cobbs, 1995]. Their disadvantage is an inherent exponential space-time barrier which usually makes them impractical. Algorithms oriented to average-case

Figure 5.1.: A fuzzy inverted list of the set of misspellings of the word *algorithm*. Each postings is a quadtruple consisting of a doc-id (first row), a word id (second row), a position-id (third row) and a score.

D9000	D9002	D9002	D9002	D9003	D9004
algorlthm	aglorithm	alorithm	alorithm	algoritm	algoritms
3	5	9	12	54	4
0.1	0.8	0.8	0.8	0.9	0.2

performance are based on filtration of potential approximate matches around the exact matches of pattern substrings by using q-gram or q-sample indexes. They perform well on average in practice as long as the error level $\delta/|P|$ is low enough [Sutinen and Tarhio, 1996b; Navarro and Baeza-Yates, 1998]. The third approach combines backtracking and exact-matching filtration. The index determines potential match areas by backtracking on short pattern pieces (factors) instead of the whole text [Navarro and Baeza-yates, 2000; Navarro *et al.*, 2000b; Kken and Na, 2007]. These *hybrid* approaches are the most promising in practice also for larger error levels. To address to large size of typical ASM indexes, various compression techniques are proposed in Russo *et al.* [2007] and Russo *et al.* [2009]. Their compressed counterparts usually have slightly worse running time but use significantly less space.

Despite this, the practicality of these indexes has been established only on relatively small texts. For example, standard test collections include sizes not larger than 64 MB [Russo *et al.*, 2007; Kken and Na, 2007; Russo *et al.*, 2009].

5.2. Algorithm Overview

Given a conjunctive query $Q = (q_1, \ldots, q_l)$, recall that to compute the set of matching documents D', the fuzzy search problem defined in Chapter 2 requires computing the intersection of union lists

$$\left(\bigcup_{j=1}^{|Q_1|} L_{w_{1,j}}\right) \cap \left(\bigcup_{j=1}^{|Q_2|} L_{w_{2,j}}\right) \cap \ldots \cap \left(\bigcup_{j=1}^{|Q_l|} L_{w_{l,j}}\right) \tag{5.1}$$

where $L_{w_{i,j}}$ is the inverted list of $w_{i,j} \in Q_i = \{w \in W \mid \mathrm{LD}(q_i, w) \leq \delta\}$, $j = 1, \ldots, |Q_i|$. Analogously, when Q is disjunctive, computing D' requires computing the corresponding merging of union lists. We will refer to the union list $L_{q_i} = \cup_j L_{w_{i,j}}$ given in sorted order as the *fuzzy inverted list* of q_i. The notion of a fuzzy inverted list will be defined more formally in the next section.

The baseline algorithm (`Baseline`) explicitly computes each L_{q_i} at query time. As pointed out in Section 2.3.1, this is (1) computationally expensive, and (2), disk I/O expensive. To compute 5.1, our algorithm tries to avoid materializing each union list L_{q_i} all along. Think of a query with $l \geq 2$ keywords, where the keywords are sorted by the length of the corresponding union-lists. Assume that we have already computed the result list $R_{q_{l-1}}$ for the first $l-1$ keywords. In summary, we compute the final result list as follows. First, we represent L_{q_l} by a small number of precomputed fuzzy inverted lists by using a fuzzy index. This addresses (2) and to some extent, also (1). Second, depending on the resulting new instance of the intersection of union list problem, we either intersect each of them with the much shorter $R_{q_{l-1}}$ which is very fast in practice; or, whenever merging is necessary, we use an optimized merging algorithm that makes uses of the Zipfian distribution of the lengths of the lists.

5.3. Fuzzy Word Index

This section is about the *fuzzy word index*, a data structure used to represent a union of a large number of inverted lists as a union of a much smaller number of precomputed lists. We will call such a union *a cover*. We start by giving a short introduction of the basic idea (Section 5.3.1) and then provide the necessary terminology and definitions (Section 5.3.2). Based on these definitions, we propose and discuss a scheme to construct our new data structure (Section 5.3.3). We then show how to compute a good cover (Section 5.3.4).

5.3.1. Introduction

The basic data structure of a fuzzy index is a *fuzzy inverted list*. Compared to an ordinary inverted list, a fuzzy inverted list corresponds not to only one but to a set of words and it comprises the list of postings for that set of words. Each word can belong to multiple sets. A *posting* in a fuzzy inverted list is a document id, word id, position, score quadruple. In each fuzzy inverted list, the postings are sorted by document id and position. The fuzzy inverted list obtained by intersection of two (or more) fuzzy inverted lists, always contains the word ids of the last fuzzy inverted list. For an example of a fuzzy inverted list see Figure 5.1. The *dictionary* of a fuzzy-word index consists of the set of all distinct words. In addition, each entry in the dictionary is equipped with pointers to the fuzzy inverted lists to which the corresponding word belongs.

Given a keyword q, we define its fuzzy inverted list L_q as the fuzzy inverted list of the set $S_q = \{w \in W \mid \mathrm{LD}(q,w) \leq \delta\}$. At query time, we would like to represent L_q as a union of a small number of precomputed fuzzy inverted lists. The basic idea behind our index is simple: instead of (contiguously) storing inverted lists of individual words, we would like to precompute and (contiguously) store the inverted lists of sets of words $s \in \mathcal{C} \subset 2^W$. More formally, we would like to compute a set $\mathcal{C}$ with $L_q \subseteq \cup_{s \in C} L_s$, where L_s is the fuzzy inverted list of s. We call the set $\mathcal{C}$ a *clustering* of W and the sets $s \in \mathcal{C}$ *clusters*. The set C is called a cover and it is defined as follows.

Definition 5.3.1 (Cover)**.** *Consider a clustering $\mathcal{C}$ of W, an arbitrary keyword q and a distance threshold δ. Let $S_q = \{w \in W \mid \mathrm{LD}(q,w) \leq \delta\}$, let L_q be the fuzzy inverted list of q and let $L(C) = \cup_{s \in C} L_s$, where $C \in \mathcal{C}$. An* exact cover *of q is any set of clusters C, with $L_q \subseteq L(C)$. An* approximate cover *of q does not necessarily contain all of L_q. We will informally say that C covers S_q or that $L(C)$ covers L_q interchangeably, depending on the current context.*

5.3.2. Properties of C and $\mathcal{C}$

Ideally, for any keyword q we would like to find a cover with $|C| = 1$ and $| \cup_{s \in C} L_s|/|L_q| = 1$. The latter is only possible if $\mathcal{C} = 2^W$, which is practically infeasible since it requires pre-computing and storing the fuzzy inverted list of every possible keyword q. In the following we define the desired properties of a cover.

Definition 5.3.2 (Properties)**.** *Given a cover $C \in \mathcal{C}$ for a keyword q, the* recall *and the* precision *of C are defined as $|S_q \cap C|/|S_q|$ and $|L_q \cap L(C)|/|L(C)|$ respectively. Furthermore, we define $|C|$ as* cover index *of C and $|L(C)|/|L_q|$ as the* processing overhead *associated with C. Finally, the* index space overhead *of $\mathcal{C}$ is defined as $Ov(\mathcal{C}) = \sum_{w \in W} \mathrm{tf}_w \cdot c_w / \sum_{w \in W} \mathrm{tf}_w$, where tf_w is the term frequency or the total number of occurrences of w in D and c_w is the number of clusters $s \in \mathcal{C}$ with $w \in s$.*

Given a distance threshold δ and a set of documents D with a dictionary W, intuitively we would like to compute a clustering $\mathcal{C}$ of W with the following properties:

- The average cover index over all distinct queries $q \in W$ is upper bounded by a value as small as possible;
- Each cover must have a given acceptable precision, recall and processing overhead;
- The index overhead of $\mathcal{C}$ must be less than a given upper bound.

In the following, we propose an intuitive and efficient clustering algorithm that achieves average precision, recall and processing overhead close to 1, cover index of 10 or less and a space overhead of about 1.5.

5.3.3. Computing a Clustering of W

Our clustering of W is based on the fact that the term frequencies in a document corpus follow Zipf's law, that is, the term frequency tf_w is inversely proportional to the rank of a given word w [Li, 1992]. In the following, it is useful to think of the frequent words as the valid words, and of the infrequent words as their spelling variants. That is usually the case in practice. However, our approach does not require this property in order to work correctly.

We make the following two observations. (i) It is natural to consider the valid words as cluster centroids of their spelling variants. (ii) The number of distinct spelling variants of a valid word depends on its frequency (more frequent valid words tend to have more spelling variants). Based on these observations, consider the following simple clustering algorithm:

Table 5.1.: The $\overline{SI}$ and $\overline{SF}$ values (see Definition 5.3.3) and corresponding percentage of rare words in the collection for different frequency thresholds t computed for the DBLP and the Wikipedia collection (Section 5.8.1).

	DBLP			WIKIPEDIA		
	$\overline{SI}$	$\overline{SF}$	% rare	$\overline{SI}$	$\overline{SF}$	% rare
$t = 250$	1.5	6.7	6.6%	6.1	22.9	3.9%
$t = 500$	1.1	5.3	8.6%	4.1	18.1	5.2%
$t = 1000$	0.9	4.4	11.3%	2.7	14.3	6.7%
$t = 2000$	0.7	3.7	15.1%	1.8	11.3	8.6%

1. Divide the words in W into a set of *frequent* and a set of *infrequent* words. We make this distinction based on a frequency threshold t that is a parameter of the algorithm;

2. For each frequent word $w \in W$, compute the set

$$s_w = \{w' \mid \mathrm{tf}_{w'} < t, \mathrm{WLD}(w, w') \leq 2 \cdot \delta\} \cup \{w'' \mid \mathrm{tf}_{w''} \geq t, \mathrm{WLD}(w, w'') \leq \delta\}$$

   and include it in C.

Definition 5.3.3. *Given a set of documents D, a distance threshold δ, and a frequency threshold t, $\overline{SI}$ is defined as the average number of frequent words with* WLD *within δ from a given infrequent word and $\overline{SF}$ is defined as the average number of frequent words with* WLD *within δ from a given frequent word.*

Table 5.1 shows the computed $\overline{SI}$ and $\overline{SF}$ values for the DBLP and Wikipedia collections for different values of the frequency threshold t.

Lemma 5.3.4. *The above clustering algorithm achieves average cover index less than $\overline{SI}$ and space overhead close to $\overline{SF}$.*

Proof. Given a keyword q, let $S_q = \{w \in W \mid \mathrm{WLD}(q, w) \leq \delta\}$. If q is a frequent word we would require only one cluster to cover S_q, namely s_q. If q is an infrequent word (possibly not in W), consider the family of sets $\{s_w\}_w \in C$, where $\mathrm{WLD}(q, w) \leq \delta$ and $\mathrm{tf}_w > t$. Let w' be a given word with $\mathrm{WLD}(q, w') \leq \delta$. If $\mathrm{tf}_{w'} \geq t$ then w' is covered by $\{s_w\}_w$ by definition. Assume $\mathrm{tf}_{w'} < t$ and let w be a word with $\mathrm{tf}_w \geq t$ and $\mathrm{WLD}(q, w) \leq \delta$. Since due to the triangle equality $\mathrm{WLD}(w, w') \leq 2 \cdot \delta$, it must hold that $w' \in s_w$. Hence, the family of sets $\{s_w\}_w$ always cover S_q.

Therefore, in average we will need less than $\overline{SI}$ clusters to cover S_q. For brevity, the second part of the proof is given in the appendix. □

Space Overhead vs. Cover Index

Let q be an arbitrary frequent word in D. Obviously, to cover S_q with a single or a small number of clusters, the space overhead $Ov(C)$ must be close to $\overline{SF}$. However, index space overhead that high might not be acceptable in practice. To reduce the space overhead at the price of a larger cover index, assume for simplicity that our clusters contain only frequent words. Since the contribution of a word w to the size of the index is $c_w \cdot \mathrm{tf}_w$, we limit the number of clusters $c_w \geq 1$ assigned to w based on tf_w. This has the effect of shrinking the clusters in C. In this regard, it is desirable words w with large $|S_w|$ to have larger cluster since their fuzzy inverted lists are associated with a higher computational cost. Alternatively, we could prefer larger clusters for words that are more likely to appear in a query. Let $1 \leq c'_w \leq c_w$ be the new number of clusters assigned to w. To this end, we assign w to its c'_w “preferred“ clusters or to its c'_w clusters with representative words that have highest likelihood to appear in a query, in case this information is available.

Let c be the average number of clusters assigned to a frequent word. The following lemma shows that the average cover index over the frequent words is now $\overline{SF} - c + 1$.

Lemma 5.3.5. *Assume that in average, each frequent word w is assigned to c clusters. Then the average cover index over the frequent words is at most $\overline{SF} - c + 1$.*

Proof. Consider the directed graph $G = (W_f, E)$ where the set of nodes is the set of frequent words W_f and $(w, w') \in E$ iff w' is assigned to the cluster s_w. Since each word is assigned to c_w clusters, $\deg^+(w) = c_w$. On the other hand, since $|s_w| = \deg^-(w)$ and the sum of the indegrees is equal to the sum of the outdegrees in G, for the average cluster size we obtain

$$\frac{1}{|W_f|} \cdot \sum_{w \in W_f} |s_w| = \frac{1}{|W_f|} \cdot \sum_{w \in W_f} c_w = c$$

Hence, given a frequent word w, if we consider all other words in s_w as singleton clusters, in average we will require at most $\overline{SF} - c + 1$ clusters to cover w. □

Remark 5.3.6. *The above upper bound limits the average cover index of all frequent words. The average cover index on the subset of preferred frequent words is less because their clusters are larger than the average cluster size c. Note also that this upper bound affects only the frequent words (its overall value is not significantly affected since the frequent words are small in number compared to $|W|$).*

Precision and Processing Overhead

Given a keyword q, let L_q be the fuzzy inverted list of S_q and let C be a cover for S_q. Observe that due to the Zipf's law, the volume in L_q mostly comes from the (few) frequent words in S_q. Based on this, we define the following two properties of a cover C.

Definition 5.3.7 (No-overlap property). *A cover C fulfills the no-overlap property, if each frequent word $w \in S_q$ is contained in a single cluster $s \in C$.*

For another desirable property of C, consider a keyword q and a cluster s_w with $\text{WLD}(q, w) \leq \delta$. For each frequent word $w' \in s_w$ (where $\text{WLD}(w, w') \leq \delta$), we would like $\text{WLD}(q, w') \leq \delta$. We would refer to this property as the *transitivity property*.

Definition 5.3.8 (Transitivity property). *A cluster s (correspondingly, an inverted list L_s) fulfills the transitivity property for a keyword q, if for each frequent word $w' \in s$, $\text{WLD}(q, w') \leq \delta$. A cover C fulfills the transitivity property for q, if each cluster $s \in C$ fulfills the transitivity property for q.*

According to our assumptions, if t is small, then a cover that fulfills the above two properties is guaranteed to have precision and processing overhead close to 1. In this context, the frequency threshold t from Table 5.1 dictates a trade-off between the cover index and the precision and processing overhead of C. For example, a very large t will imply smaller average cover index but also clusters with lower precision and higher processing overhead.

None of the above two properties are fulfilled in general. To see this, consider an infrequent keyword q and a cluster s_w with $\text{WLD}(q, w) \leq \delta$ and let $w' \in s_w$ be a frequent word. Due to the triangle inequality, $\text{WLD}(q, w')$ can be as large as $2 \cdot \delta$. We address this as follows. We split each cluster $s \in C$ into two separate clusters: a cluster with frequent and a cluster with infrequent words. A cluster with infrequent words always satisfies the transitivity property. A cluster s with frequent words may be included in C only if $s \subseteq S_q$. This increases the upper bound of the average cover index over the infrequent words by a factor of 2.

A non-overlapping clustering of W is a clustering where $c_w = 1$ for each frequent word $w \in W$. If C is a non-overlapping clustering, then obviously the no-overlap property is always fulfilled. In contrast, if C is not non-overlapping, nothing prevents a cover to contain two different clusters with a frequent word in common. Hence, we must enforce the no-overlap property by considering only clusters in C that do not have a frequent word in common. While this might affect the average cover index, it does not affect its upper bound.

5.3.4. Computing an Optimal Cover

Given a clustering C and a keyword q, a cover for q that fulfills the transitivity and the no-overlap property and has a minimal cover index is called optimal. The clustering scheme from the previous section provides only an upper bound on the average cover index. This is because the scheme considers only a restricted set of clusters. Note that to find an optimal cover by exhaustive search would be too expensive, compared to the total query

Table 5.2.: Average number of similar words, average cover index and space overhead.

	DBLP	Wikipedia
Average number of similar words	132	251
Average cover index	5	10
Space overhead	1.4x	1.5x

processing time. This is because each word in S_q can belong to multiple clusters. The number of relevant clusters in practice typically varies from few hundreds to few thousands. Instead, we employ a greedy heuristic as follows.

We first impose the following rules

- Let I be the set of already covered words from S_q and let s be a given cluster with frequent words. Then s can be included in a cover C if $s \subseteq S_q - I$;
- If s is a cluster with infrequent words, then s can be included in C if it contains at least $K \geq 2$ words from S_q that have not been covered before;
- If there is no such cluster, then each word is considered as a singleton cluster.

The problem of computing an optimal cover now reduces to finding a cover with minimal cover index. This is an optimization problem similar to the *set cover problem*. Given a set $\mathcal{U}$ and n other sets whose union comprises $\mathcal{U}$, the set cover problem is to compute the smallest number of sets whose union contains all elements in $\mathcal{U}$. In our version of the problem, however, there is a dependency that some pairs of sets cannot be chosen by the algorithm simultaneously. The set cover problem is NP-complete. We use the following greedy algorithm that has been shown to achieve a logarithmic approximation ratio [Lund and Yannakakis, 1993].

1. Compute S_q and consider all clusters $s \in C$ that contain at least K words from S_q;
2. Pick the cluster s that contains the largest number of uncovered words in S_q, preferring smaller clusters in the case of ties, and include s in C;
3. Take the covered words out of consideration and iterate if S_q is not yet covered.

Table 5.2 shows the average cover index and space overhead achieved on two of our test collections.

5.4. Fuzzy Prefix Index

This subsection is about the fuzzy prefix index, a data structure analogous to the fuzzy word index from the previous section. We start by giving a short introduction to the problem (Section 5.4.1) and propose a precomputation algorithm similar to that from Section 5.3.3. We provide evidence why the same algorithm is less effective when applied to prefixes (Section 5.4.2). We then propose a different method for pre-computing fuzzy inverted lists based on prefixes with “don’t care“ characters, that by design fulfill the transitivity property (Section 5.4.3). As before, at the end of the section we show how to compute a good cover by using our new index (Sections 5.4.4 and 5.4.5).

5.4.1. Introduction

The fuzzy prefix index is a data structure that can represent the fuzzy inverted list L_q of a given prefix q as a union of a small number of precomputed fuzzy inverted lists. The difference to the fuzzy word index is that q is a prefix and that prefix instead of word Levenshtein distance is used. As before, given an acceptable index space overhead, the problem is to precompute a set of fuzzy inverted lists so that at query time, a cover of S_q can be computed with favorable precision, recall and processing overhead (defined as before). However, the problem is more challenging when dealing with prefixes and prefix Levenshtein distance since the size of S_q is very different for different prefix lengths $|q|$.

5.4.2. Clustering Prefixes

We first propose an approach that is analogous to the word clustering algorithm from the previous section. Given a prefix p of some predefined length, the term frequency of p is defined as $\text{tf}_p = \sum_{w, p \leq w} \text{tf}_w$, where $w \in W$. As before, we set a threshold t for frequent prefixes and for each frequent prefix p compute the fuzzy inverted lists $\cup_{p' \in s_p} L_{p'}$ and $\cup_{p' \in s_p'} L_{p'}$, where

$$s_p = \{p' \mid \text{tf}_{p'} \geq t, \text{PLD}(p, p') \leq \delta\} \text{ and } s_p^{'} = \{p' \mid \text{tf}_{p'} < t, \text{PLD}(p, p') \leq 2 \cdot \delta\}$$

An important observation we made used of in the clustering from the previous section was that $\overline{SI}$, the upper bound on the average cover index, was relatively small in practice. However, this does not apply in case of prefixes. This is because, first, many frequent words with WLD above the threshold will have PLD below the threshold because they will have equal (or similar) prefixes. Second, many frequent and infrequent words have equal prefixes and will be considered as a single (frequent) prefix. In addition, this makes the transitivity problem more severe than before because large fraction of the prefixes will be frequent.

5.4.3. Prefixes with “don’t care“ Characters

We simplify the problem by assuming that the query length $|q|$ is fixed to k characters, where k is small. Instead of words, we speak in terms of covering the set of k-prefixes in W_k. Later we show how to extend the solution to the problem from queries of length k to queries of any length.

To tackle the transitivity problem, we introduce the notion of prefixes with “don’t care“ characters. For example, consider the prefix p=`al*o`, where the star can match any single character from the alphabet. If q is a completion of a prefix with “don’t care characters“ p, we will write $q \preceq_* p$. For example, `algorithm` $\preceq_*$ `al*orithm`. The fuzzy inverted list of p is simply the union of the inverted lists of all words in W that contain a prefix that matches p. For brevity, we will refer to prefixes with “don’t care“ characters as $*$-prefixes. Our fuzzy prefix index will be based on indexing $*$-prefixes. In the following we show how to compute a set of $*$-prefixes to cover the set S_q for a given prefix q by preserving the transitivity property.

Lemma 5.4.1. *Given a threshold δ, let p be a prefix with δ “don’t care“ characters and length k and let $S_p = \{w \in W \mid w[k] \preceq_* p\}$. If q is a prefix that matches p, then $\forall w \in S_p, \text{PLD}(q, w) \leq \delta$.*

Proof. The δ “don’t care“ characters correspond to δ substitution errors on fixed positions for each $p' \in S_p$. Hence $\text{WLD}(q, w[k]) \leq \delta$. □

Given a prefix $q = p_1 p_2 \ldots p_m$, obviously the set of prefixes $p_1' p_2' \ldots p_n'$, where $p_i' = p_i$ for $i \neq j$ and $p_i' = *$ for $i = j$, where $j = 1 \ldots n$, will cover (at least) the prefixes that contain a single substitution error. However, it is not immediately clear how to deal with prefixes that contain deletion or insertion errors.

Example 5.4.2. *Consider the prefix* `algo` *and the 4 $*$-prefixes:* `*lgo`, `a*go`, `al*o` *and* `alg*`. *Consider the prefix* `agor` *(obtained from* `algorithm` *by a deletion at position 2). Observe that this prefix does not match any of the 4 $*$-prefixes.*

In the following lemma we show hwo to match prefixes with deletion and/or insertion errors.

Lemma 5.4.3. *Given a threshold δ and a prefix q, let W_k^* be the set of all $*$-prefixes with s “don’t care“ characters generated from W_k, where s is an arbitrary but fixed number from the interval $1 \ldots \lceil \delta/2 \rceil$. Let W_k^{**}, in addition to W_k^*, contains all $*$-prefixes of length $k + \delta$ in which the “don’t care“ characters can appear only in the first k positions. Let C be the set of all prefixes $p \in W_k^{**}$ with $\text{PLD}(q, p) \leq \delta$. Then $\cup_{p' \in C} L_{p'}$ covers L_q and $\text{PLD}(q, w) \leq \delta$, for any $w \in W$ such that $\exists p \in C, p \preceq_* w$ (i.e., the transitivity property is fulfilled).*

Proof. The proof is based on showing that there is a family of $*$-prefixes in W_k^{**} that covers each error type (insertions, deletions and substitutions) as well as their combinations. A detailed version is given in the appendix. □

Remark 5.4.4. *Note that it is not necessary to limit s to $\lceil \delta/2 \rceil$ as in Lemma 5.4.3, however, certain prefixes might then not be covered due to the subtlety explained at the end of the proof. This can be circumvented if we include the $*$-prefixes with s' “don’t care“ characters for a fixed $s' \leq \lceil \delta/2 \rceil$, in addition to the existing $*$-prefixes.*

Example 5.4.5. *Consider the prefix* `algo` *and assume* $\delta = 1$. *Consider the following sets of prefixes*

```
*lgo   *lgor   alg*
a*go   a*gor
al*o   al*or
alg*   alg*r
```

Obviously the first set of prefixes covers substitution errors in `algo`. *The second set of prefixes covers deletion errors in* `algo`. *Assume there is a deletion in the second position, resulting in the prefix* `agor`. *Then words starting with* `algor`, *like* `algorithm` *(or with any other prefix of the form* `a*gor`*) are covered by* `a*gor` *and* $\mathrm{PLD}(\texttt{agor}, \texttt{a*lgor}) \le 1$. *Now assume there is an insertion error in* `algo` *at position 2, resulting in the prefix* `axlg`. *Then* `algo` *will be covered by the prefix* `alg*` *and* $\mathrm{PLD}(\texttt{axlg}, \texttt{alg*}) \le 1$. *Note that the* $*$*-prefixes covering insertion errors (e.g.* `alg*` *above) are already included by the set of* $*$*-prefixes covering substitution errors.*

Space Overhead vs. Cover Index

The space overhead of the fuzzy prefix index is defined analogously as the space overhead of the fuzzy word index. If the $*$-prefixes are regarded as sets of prefixes, the space overhead of the index is mainly determined by the average number of different $*$-prefixes containing a given frequent prefix p. The number of "don't care" characters s in Lemma 5.4.3 provides a trade-off between the cover index (the total number of $*$-prefixes needed to cover S_q) and the space overhead of the index. Prefixes with more "don't care" characters are able to simultaneously cover larger set of prefixes with PLD within δ from q, however, they require more space when included in the index. For example, the space overhead in the example set of prefixes provided above is at most 8x (at most 4x for the first set and at most 4x for the second set of $*$-prefixes). Note that each 4-prefix in W should be matched by (included in the fuzzy inverted list of) only one $*$-prefix per set.

As before, to limit the size of the index, we could assign each frequent prefix p only to a limited number of $*$-prefixes. In other words, the fuzzy inverted list of p will not be necessarily contained in the fuzzy inverted list of each $*$-prefix that matches p.

5.4.4. Computing an Optimal Cover

Given a keyword q, a cover computed according to Lemma 5.4.3 is guaranteed to cover S_q and simultaneously fulfill the transitivity property. However, such a cover is not necessarily optimal. Namely, since many words in W are covered by multiple $*$-prefixes, not all $p \in W_k^{**}$ with $\mathrm{PLD}(q, p) \le \delta$ are always required. Hence, we still have to compute a minimal set of $*$-prefixes that cover S_q.

Optimal, is the cover with minimal cover index such that the no-overlap property is fulfilled. The problem of computing an optimal cover is similar to the set cover problem discussed in the previous section. We employed a greedy heuristic similar to that from Section 5.3.4.

Table 5.4 and Table 5.5 show the average cover index and space overhead for different limits on the number of $*$-prefixes assigned to the frequent prefixes. As a reference, Table 5.3 shows the average cover index achieved without imposing a limit neither on the size of the index nor on the processing overhead of the computed covers.

5.4.5. Computing a Cover for an Arbitrary Keyword Length

If the new keyword is obtained by adding a letter to the old keyword, in Section 5.5.1 we show how the new query result can be computed incrementally from the old query result. Hence, computing a cover for the new keyword is not required. Assume the opposite and recall that our index contains only $*$-prefixes of length k. To be able to obtain a cover for a keyword q with length different than k we could index $*$-prefixes with multiple lengths. However, this would require too much space. Instead, we opt for an index with short $*$-prefixes of fixed length k. To obtain a cover for a keyword q of arbitrary length, we first compute a cover C by using the $*$-prefixes of length k. Note that when $|q| > k$ it is likely that we would need less $*$-prefixes to compute a cover for q compared to when $|q| = k$. Then we filter C to produce a refined cover C' that fulfills the transitivity property.

Table 5.3.: Average cover index and space overhead by using prefixes with “don’t care“ characters to cover the fuzzy inverted list of a prefix without limitation neither on the space overhead nor on the processing overhead.

	DBLP		Wikipedia	
	Avg. cover index	Space overhead	Avg. cover index	Space overhead
$\delta = 1$	5.7	3.1x	5.6	3.5x
$\delta = 2$	21.1	12.5x	37.5	14.0x

Table 5.4.: Average cover index for prefixes of length $k = 4$ and threshold $\delta = 1$ for different space overheads by limiting the maximum number of prefixes with “don’t care“ characters per k-prefix.

	DBLP		Wikipedia	
Limit	Avg. cover index	Space overhead	Avg. cover index	Space overhead
1	9.9	0.7x	18.0	1.1x
2	8.5	1.1x	14.2	1.6x
3	8.0	1.4x	12.9	2.1x
∞	7.7	1.7x	12.8	2.8x

Suppose that $L_1, \ldots, L_m$ are the fuzzy inverted list of the clusters in C. We hash the word-ids in S_q or store them in a bit-set and then compute L'_i by iterating through L_i and appending the postings with word-ids in S_q by performing a hash or bit-set look-up. The lists $L'_1, \ldots, L'_m$ are then the refined cover for q. We note that the filtering procedure typically takes time negligible compared to the total query processing time.

5.5. Efficient Intersection of Union Lists

In this section, we show how to efficiently compute the intersection of union lists once they have been obtained from a fuzzy index. We first make use of the skewed distribution of list lengths by employing a known greedy merging technique (Section 5.5.2). We then show how the intersection of union lists can be computed more efficiently depending on the particular problem instance (Section 5.5.1).

Recall that L_{q_i} is the fuzzy inverted list of the i-th keyword given as a union list in Equation 5.1. By using a fuzzy index, each L_{q_i} is represented by a small number of precomputed lists. As already mentioned, this has two advantages. First, the union lists are already partially merged, and, second, fetching the lists from disk requires less disk I/O. Apart from this, we will show that the intersection of union list problem can now be computed more efficiently. Let Q be a query with l keywords (sorted in increasing order by the corresponding union list lengths) and assume that Q is being processed from left to right. Let R_{q_i} be the result of intersecting the first i union lists and assume that we have already computed $R_{q_{l-1}}$. What remains to be computed is

$$R_{q_{l-1}} \cap \left(\bigcup_{j=1}^{n_l} L_{q_l}^j \right) \tag{5.2}$$

where $L_{q_l}^j$, $j = 1 \ldots n_l$ is a cover of L_{q_l}.

5.5.1. Two Variants for the Intersection of Union Lists

By the law of distributivity, the final result list R_{q_l} can be obtained either by computing the left-hand side of Equation 5.3 (for brevity, we will refer to this as *variant 1* of processing) or by computing its right-hand side (*variant 2* of processing):

$$R_{q_{l-1}} \cap \left(\bigcup_{j=1}^{n_l} L_{q_l}^j \right) = \bigcup_{j=1}^{n_l} \left(R_{q_{l-1}} \cap L_{q_l}^j \right) \tag{5.3}$$

Table 5.5.: Average cover index for prefixes of length $k = 6$ and threshold $\delta = 2$ for different space overheads by limiting the maximum number of prefixes with “don’t care“ characters per k-prefix.

	DBLP		Wikipedia	
Limit	Avg. cover index	Space overhead	Avg. cover index	Space overhead
1	19.2	0.7x	35.9	2.1x
2	18.4	1.0x	31.4	2.4x
3	17.9	1.2x	29.2	2.7x
∞	16.8	2.3x	26.2	5.0x

It turns out that variant 2 can be computed significantly faster than variant 1 (and vice-versa), depending on the problem instance. Too see why, we will consider the dominating cost to compute the left and right-hand side of Equation 5.3. The cost of variant 1 is roughly equal to the cost to merge the l-th union lists. The cost of variant 2 is roughly equal to the cost to intersect $R_{q_{l-1}}$ with each of the fuzzy inverted lists $L^j_{q_l}$, $j = 1, \ldots, n_l$. Under the assumption that n_l is small, this cost depends merely on the length of $R_{q_{l-1}}$ and therefore it will reduce when the length of $R_{q_{l-1}}$ reduces. However, the cost of variant 1 will remain roughly the same. Therefore, for sufficiently small $R_{q_{l-1}}$, the cost of variant 2 will be much smaller than the cost of variant 1. Note that our fuzzy indexes take care that n_l is usually small in practice. Since the union lists are sorted by their lengths in increasing order and since $R_{q_{l-1}}$ is obtained by intersecting $R_{q_{l-2}}$ with $L_{q_{i-1}}$, $|R_{q_{l-1}}|$ is always much less than $|L_{q_l}|$ when the query has more than 2 keywords.

To see more formally when the cost of variant 2 is less than the cost of variant 1, let n_1 be the length of $R_{q_{l-1}}$, n_2 be the length of L_{q_l}, $n_{2,i}$ the length of $L^i_{q_l}$ and M the length of R_{q_l}. Let $c_I \cdot (n_1 + n_2)$ and $c_M \cdot (n_1 + n_2)$ be the running times to intersect and merge two lists with lengths n_1 and n_2 respectively. The running time of variant 1 then is

$$c_M \cdot \log n_l \cdot n_1 + c_I \cdot (n_1 + n_2) \tag{5.4}$$

while the running time of variant 2 is

$$c_I \cdot \sum_{i=1}^{n'_l} (n_1 + n_{2,i}) + c_M \cdot \log n_l \cdot M \tag{5.5}$$

$$= c_I \cdot n'_l \cdot n_1 + c_I \cdot n_2 + c_M \cdot \log n_l \cdot M \tag{5.6}$$

We opt for linear list intersection unless the length difference of the lists is extreme. Thanks to its perfect locality of reference and compact code, linear list intersection is often a faster option in practice compared to other asymptotically more efficient algorithms based on binary searches [Demaine *et al.*, 2000; Baeza-Yates, 2004]. From Equations 5.4 and 5.5, we get that variant 2 is faster than variant 1 as long as

$$n_l < 1 + \frac{n_2}{n_1} \cdot \frac{c_M}{c_I} \cdot \log n_l$$

First, note that $c_I < c_M$ since merging is associated with much larger constant factor than list interesection. Second, n_1 is typically much smaller than n_2 for $l \geq 2$. Since n_l is usually small, the above inequality is typically satisfied. Hence, in most of the cases, only the first (the shortest) union list must be fully materialized. This results in large advantage of variant 2 over variant 1 when the difference among the union list sizes is large (see Figure 5.2, left). If the inequality is not satisfied, then we resort to variant 1. In the next section, we show how to merge the fuzzy inverted lists faster, whenever we have to, by making use of the Zipfian distribution of their lengths.

It should be noted that the advantage of our approach over `Baseline` increases with the number of keywords in Q. To see this, suppose we have queries with l (uniformly distributed) keywords. Assume that C^o_i and C^b_i for $i = 1, \ldots, l$ are costs associated with processing the i-th keyword for `Baseline` and our approach respectively. Due to the linearity of the means, the average advantage over `Baseline` on l-word queries can be written as

$$\frac{\sum_{i=1}^{l} C^b_i}{\sum_{i=1}^{l} C^o_i} \tag{5.7}$$

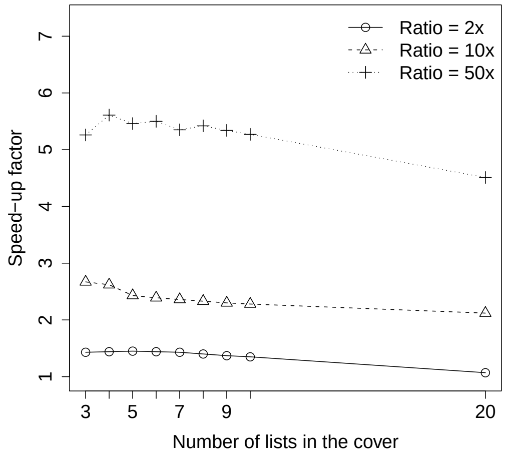


Figure 5.2.: Advantage of variant 2 over variant 1 of processing the intersection of union lists problem for different number of lists and different ratios between the list sizes (see Section 5.5.1).

This would simply be equal to C_1^b/C_1^o (the advantage on one-word queries) if the costs associated with q_i are independent of i. This is indeed the case for `Baseline` since C_i^b is dominated by the cost of multi-way merge which, by assumption, is independent of i. For our approach, however, C_i^o decreases with i since it is typically dominated by the cost to intersect the result list up to the $i-1$th keyword (whose length decreases rapidly) with a union list. Hence, (5.7) decreases with l.

5.5.2. Merging Lists of Skewed Lengths

The standard way to merge l lists of similar sizes is to maintaining a priority queue for the frontier elements of the lists. This procedure is known as multi-way merge. Multi-way merge is optimal as long as the lists have similar lengths. The following example provides the intuition why.

Example 5.5.1. *Say we would like to merge 1,000 lists with a total of 100,000 elements using a binary min-heap. Say the first 5 lists contain the smallest 95% of all elements. Observe that during the merging of these lists, the frontier elements of the remaining 995 lists will remain idle in the priority queue for 95% of the calls of the delete-min operation.*

If the lists are of widely different lengths, it is intuitive that one could make a better use of the distribution of list lengths by merging lists of similar lengths first. We make use of the observation that the length distribution of the fuzzy inverted lists in a cover is more skewed than the length distribution of the inverted lists of the individual words covered. Assume for a moment that we restrict ourselves to binary merges. Then any merging algorithm defines a merge tree, where the leaves correspond to the initial lists and the root correspond to the final merged list. The depth of each leaf is equal to the number of merging steps of the corresponding list. The total merging cost is hence given by $\sum_i d_i \cdot l_i$, where d_i is the depth of the leaf corresponding to the ith list and l_i is the length of the ith list. It is well known that the optimal sequence of merges, i.e., the one that minimizes the cost, is obtained by repeatedly merging the two lists of shortest lengths. This can be easily seen by resorting to Huffman coding [Huffman, 1952] as the two problems have equivalent definitions. The merge tree is then identical to the code tree produced by the Huffman algorithm. We will refer to this algorithm as *optimal binary merge*.

The running time of optimal binary merge to merge m lists with n elements in total is proportional to $K \cdot n$, where K depends on the distribution of list lengths (as well as on m). For certain distributions, K can be computed exactly. For example, if the lists have equal lengths, then $K = \log m$, i.e., we obtain the running time of multi-way merge. This can be seen by observing that the merge tree is a full binary tree. Hence, in general, $K \leq \log m$. If the length of the ith longest list is proportional to $1/2^i$, then it is not hard to show that $K \leq 2$. Figure 5.3 (left) compares the empirically computed K values for the Zipfian and the uniform distribution.

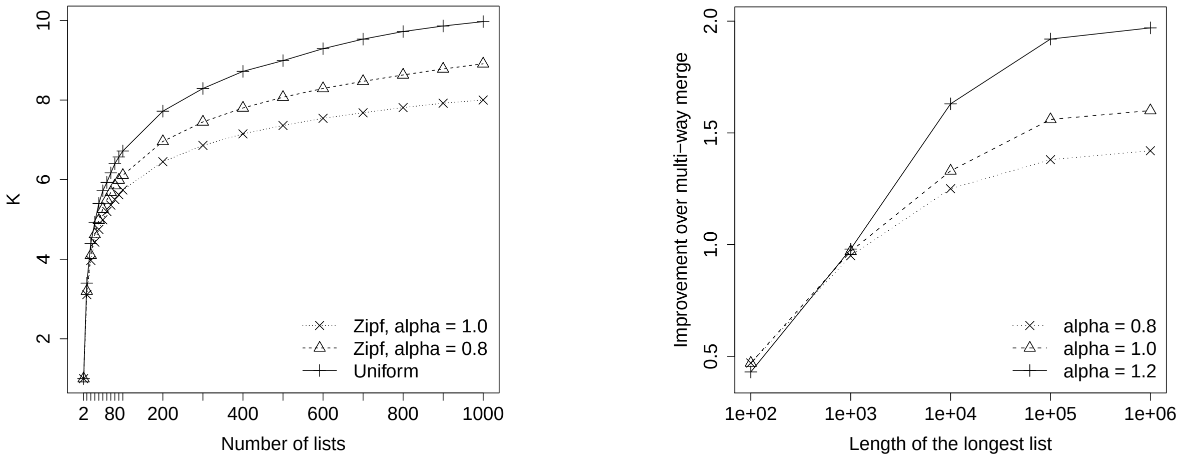


Figure 5.3.: Left: The K value (defined in Section 5.5.2) for uniform and Zipfian distribution. Right: Advantage of optimal binary merge over multi-way merge on lists with lengths exhibiting various Zipfian distributions given as a ratio of their running times.

The idea can be generalized by simultaneously merging the next k shortest lists by using k-way merge. Binary merging, however, gave the best results in practice. This is because the simple and compact algorithm for merging two lists has a much smaller constant factor in its running time compared to k-way merge. Figure 5.3 (right) shows the advantage of optimal binary merging over multi-way merge given as a ratio of their running times. The distribution of the list lengths was Zipfian. Multi-way merge was faster in practice when the length of the longest list was relatively small due to the long tail of the Zipfian distribution. Optimal binary merge in this scenario has a larger overhead per element since it has to repeatedly merge two very short lists (e.g. lists of length 1). Multi-way merge, on the other hand, always requires the same amount of work per element, i.e., a single delete-min operation.

5.6. Efficient Merging of Union Lists

It is straightforward how to use a fuzzy index for efficient merging of union lists. After representing each union list in partially merged form, the merging of union lists problem is given by

$$\left(\bigcup_{j=1}^{n_1} L_{q_1}^j\right) \cup \left(\bigcup_{j=1}^{n_2} L_{q_2}^j\right) \cup \ldots \cup \left(\bigcup_{j=1}^{n_l} L_{q_l}^j\right)$$

where $L_{q_i}^j$, $j = 1 \ldots n_i$ is a cover of L_{q_i}. The short fuzzy inverted lists are merged by using multi-way merged and the rest are merged by using optimal binary merge as done in Section 5.5.2. An additional optimization that could make a better use of the skewed distribution of list lengths is to consider all union lists at once and merge them simultaneously, rather than merging them one by one.

5.7. Cache-based Query Processing

If an old query is a prefix of a newly typed query (e.g., when the user types the query letter by letter), the intersection or merging of union lists can be computed incrementally by using previously cached results in time negligible compared to computing the result from scratch. Suppose the user has typed a query Q_1 for the first time. The result list for Q_1 is then computed from scratch and stored in memory. Suppose also that Q_1 is a prefix of a new query Q_2. Recall that the query processing is done from left to right which in turn means that the last computed result list contains the word ids of the union list of the last keyword. There are three cases to be considered:

Table 5.6.: Summary of the test collections used in the experiments.

Collection	Raw size	Documents	Occurrences	Dictionary size
DBLP	1.1 GB	0.3 millions	0.15 billions	1.2 millions
Wikipedia	21 GB	9.3 millions	3.2 billions	29 millions
GOV2	426 GB	25.2 millions	23.4 billions	60 millions

1. The number of keywords in Q_1 and Q_2 is equal (the last keyword in Q_1 is a prefix of the last keyword in Q_2), for example, `inform` and `informa`. The result of Q_2 in this case can be computed merely by filtering the result list for Q_1 by hashing word-ids, a procedure that has been already described at the end of Section 5.4.5;

2. Q_2 includes additional keywords and the last keyword in Q_1 has equal length to the corresponding keyword in Q_2, for example, `information` and `information retr`. The result list of Q_1 in this case is simply reused as already computed partial result and the intersection / merging of union lists is carried out as usual;

3. Q_2 includes additional keywords and the last keyword in Q_1 is shorter than the corresponding keyword in Q_2, for example, `inform` and `information retr`. In this case the result list of Q_1 is initially filtered as in (1) and then reused as in (2).

5.8. Experiments

In this section, we present the experimental results for our core fuzzy search algorithms introduced in this chapter. For completeness, we include the running times required to compute the fuzzy suggestions of a given query, although the algorithm is presented in the next chapter.

We start by giving overview of our test collections (Section 5.8.1) and then present experimental results for fuzzy keyword-based search (Section 5.8.2), followed by experimental results for fuzzy prefix search (Section 5.8.3).

We have implemented all our algorithms and evaluated them on various data sets. All our code is written in C++ and compiled with GCC 4.1.2 with the -O6 flag. The experiments were performed on a single core. The machine was Intel(R) Xeon(R) model X5560 @ 2.80GHz CPU with 1 MB cache and 30 GB of RAM using a RAID file system with sequential read/write rate of up to 500 MiB/s. The operating system was Ubuntu 10.04.2 in 64-bit mode.

5.8.1. Test Collections

Our experiments were carried out on three test collections of various sizes:

DBLP: a selection of 31,211 computer science articles with a raw size of 1.3 GB; 157 million word occurrences (5,030 occurrences per document in average) and 1.3 million distinct words containing many misspellings and OCR errors;

WIKIPEDIA: a dump of the articles of English Wikipedia with a raw size of 21 GB; 9,326,911 documents; 3.2 billion word occurrences (343 occurrences per document in average) and a diverse dictionary of 29M distinct words with an abundance in foreign words;

GOV2: the TREC Terabyte collection with a raw size (including html tags) of 426 GB of which 132 GB are text. It contains 25,204,103 documents; 23.4 billion word occurrences (930 occurrences per document in average) and around 60 million distinct words. The goal was to investigate how well our fuzzy search scales on a larger collection.

5.8.2. Fuzzy Word Search

In this section, we evaluate the performance of our fuzzy keyword-based search. The algorithms were tested by computing the full result lists on a single core, without employing top-k processing heuristics or other thresholding based heuristics for early termination.

Compared Algorithms

We evaluated the following algorithms:

1. The `Baseline` algorithm given at the beginning of Section 2.3. `Baseline` does not require additional space because employs only an inverted index;

2. Our proposed method based on a fuzzy word index (Sections 5.3 and 5.5). The index size overhead was within a factor of 2 on all test collections;

3. The `ForwardList` algorithm from Ji *et al.* [2009] described in Section 5.1. The index size overhead was equal to roughly a factor 2 on all test collections. This is because the forward lists (computed for each document) in total require space equal to that of an uncompressed inverted index (it is not clear how to effectively compress the forward lists as compression is not discussed in the original article);

4. The fuzzy search from the state-of-the-art open-source search engine Lucene (version 3.3) which we include as a reference. Similarly as `Baseline`, Lucene does not require additional space.

We want to stress that we took special care to implement `Baseline` and `ForwardList` efficiently. In particular, `Baseline` decided between variant 1 and variant 2 of processing the intersection of union list depending on the number of lists (see Section 5.5). Also, we insured that the inverted lists are laid out on disk and processed in lexicographic order to minimize disk I/O (see Section 2.3.1). As for `ForwardList`, we employed optimizations proposed by the original authors but also by ourselves. For example, we precomputed the size of each possible union list corresponding to a prefix in advance so that the shortest union list is known exactly instead of being guessed by estimating the lengths as proposed in Ji *et al.* [2009]. Furthermore, we implemented an alternative version of the original algorithm based on materializing the shortest union list first and then removing the documents with forward lists without word ids in the required word ranges. This improved the running time of the original algorithm on fuzzy search for up to a factor of 4 on small collections and up to a factor of 2 on larger collections.

To compute the set of words (prefixes) similar to a given keyword, we integrated the `DeleteMatch` algorithm (Section 3.2) with our approach as well as with `Baseline` using a dynamic distance threshold. We integrated `PrefixTrie` with the `ForwardList` algorithm and set the distance threshold to 1.

Queries

We experimented on 200 real two-word queries that did not contain errors. Then we applied from 0 to 3 edits to each keyword as described in Section 3.4.2. For each query, we measured the time associated with in-memory processing, the time needed for disk I/O and the time to compute the top-10 query suggestions. We carried out the experiments on a positional and a non-positional index (whenever possible).

Discussion

Table 5.7 shows the average running time to compute the intersection of union lists on two-word queries. The time required to decompress the fuzzy inverted lists (which is not included) required around 40% of the total in-memory running time on DBLP and around 30% of the total in-memory running time on Wikipedia and GOV2. Table 5.8 shows the average query processing time when the index resides on disk.[1] The time required to decompress the fuzzy inverted lists required around 8% of the total running time on DBLP and below 5% on Wikipedia and GOV2. The following is a summary of the results:

[1] The *ForwardList* algorithm is based on in-memory non-positional index.

Table 5.7.: Average running time required for the intersection of union-lists on two-word queries (the index of *ForwardList* on the GOV2 collections was too big to fit in memory).

	DBLP	Wikipedia		GOV2	
	non-positional	non-positional	positional	non-positional	positional
ForwardList	9.4 ms	296 ms	-	-	-
Baseline	9.1 ms	54 ms	155 ms	460 ms	1,044 ms
Ours	1.6 ms	7.2 ms	19 ms	45 ms	133 ms

Table 5.8.: Total average fuzzy keyword-based search query processing times on two-word queries when the index is on disk.

	DBLP	Wikipedia		GOV2	
	positional	non-positional	positional	non-positional	positional
Apache Lucene	1,461 ms	-	28,073 ms	-	52,846 ms
Baseline	61 ms	1,431 ms	2,240 ms	4,597 ms	6,543 ms
Ours	30 ms	219 ms	638 ms	517 ms	1,865 ms

- Our algorithm improves `Baseline` for up to factor of 9 on two-word queries and for up to a factor of 12 on three-word queries (results are not shown) when the index is in memory;

- The advantage of our algorithm increases with the size of the collection and it is larger when the index resides in memory;

- The most expensive (yet necessary) part of the algorithm when the index resides on disk is reading large volumes of data.

The `ForwardList` algorithm was fast when the text collection was small or when exact (prefix) search was employed. When it comes to fuzzy search on comparatively larger test collections, it turned out that `ForwardList` is less efficient than our `Baseline` algorithm. This is because the set of words similar to a keyword does not correspond to a single word range, a key property utilized by the algorithm. In the worst case, a binary search must be performed for each word in this set separately. Hence, `ForwardList` does not scale logarithmically with respect to the number of similar words but linearly instead (note that this is more pronounced on fuzzy keyword search than on fuzzy prefix search). On two-word queries its worst-case running time is proportional to $m \cdot N \cdot \log_2 k$, where m is the number of distinct similar words, k is the average number of distinct words per document and N is the total volume of the shorter union list. In contrast, the running time of `Baseline` is proportional to $2(N \cdot \log_2 m + N)$. Hence, the number of distinct similar words (or different inverted lists) has therefore only a logarithmic impact.

Table 5.8 includes Lucene's average query processing times on fuzzy search queries. Although we expected running times similar to that of `Baseline`, Lucene's performed substantially worse. One reason for this is the straightforward implementation of fuzzy word matching in the version that we tested.

Table 5.9 shows a break-down of the total fuzzy search query processing time in three categories: in-memory processing, query suggestion and disk I/O. Clearly, the most expensive part of the query processing was reading large volume of data. This becomes more severe for longer queries. Hence, any approach that needs to read the (full) fuzzy inverted list of each keyword would be problematic in practice when the query is long. Therefore, an important and interesting future research direction is approach that does not have this requirement (if such an approach exists at all).

5.8.3. Fuzzy Prefix Search

In this section, we evaluate the performance of our fuzzy prefix search. As before, each algorithm computed the full result lists, without employing heuristics for early termination.

Table 5.9.: Break-down of the total fuzzy keyword-based search query processing time on Wikipedia.

In-Memory Query Processing	Query Suggestion	Disk I/O
22%	10%	68%

Compared Algorithms

We evaluate the same algorithms from the previous section. Since Lucene currently does not support fuzzy prefix search, as a reference we included an exact prefix search realized by using an inverted index.

We constructed a fuzzy prefix index (see Section 5.4) by using $*$-prefixes of length 4, with a single "don't care" character ($k = 4$ and $s = 1$ in Lemma 5.4.3). This allows a single error in the first 4 characters of a keyword, but has the advantage of a smaller index (slightly larger than a factor of 2 on all test collections), less irrelevant results as well as avoiding full re-computation of the query when the last keyword is being typed letter by letter.

We used the `DeleteMatchPrefix` algorithm from Section 4.2 to compute the set of similar completions for our approach and `Baseline` by using a dynamic distance threshold; and `PrefixTree` for `ForwardList` with distance threshold set to 1.

Queries

We experimented with 200 real two-word queries that did not contain errors and then applied a random number of errors to each keyword as described in Section 3.4.2. Then we "typed" the last keyword letter by letter, with a minimal prefix length of 4. For example, the query `corrupted politician` gives rise to 7 queries: `corrupted poli`, `corrupted polit`, `corrupted politi`, etc. As before, for each query we measured the time required for disk and in-memory query processing and the time to compute the top-10 query suggestions.

Discussion

Table 5.10 shows the average running time to compute the intersection of union lists on two-word queries and Table 5.11 shows the average query processing times when the index resides on disk. As before, the time required to decompress the fuzzy inverted lists required around 40% of the total running time on DBLP and around 30% of the total running time on Wikipedia and GOV2 when the index resides in memory; and below 10% of the total running time on DBLP and below 5% of the total running time on Wikipedia and GOV2 when the index resides on disk. The following summarizes the results:

- Our algorithm outperforms `Baseline` for up to a factor 7 on two-word queries and up to a factor of 9 on three-word queries (results are not shown) when the index is in memory and up to a factor 4 when the index resided on disk (the advantage is larger on short prefixes). The advantage increases when the number of query words increases (as predicted at the end of Section 5.5) or when the disk is slow;

- Our algorithm achieves running times similar to those when exact prefix search is used with an inverted index when the index resides in-memory;

- As before, most expensive part of the algorithm is reading large volumes of data from disk;

- Computing the result incrementally by using caching reduces the running time dramatically.

The `ForwardList` algorithm performed faster on fuzzy prefix search compared to fuzzy keyword search but it was still slow on larger collections for reasons already elaborated in the previous section. Nevertheless, if the index resides on disk, an advantage of `ForwardList` is that only the first (or the shortest) inverted fuzzy list needs to be fetched from disk. Therefore, this algorithm may benefit when the query consists of a large number of keywords. It is not clear how to extend this algorithm to work with a positional index.

The difference in the running times was smaller when the index resides on disk due to the large amount of volume that both algorithms have to read from disk. Table 5.12 shows that this is the most expensive part of the query processing. For example, 130 MB per query were read from disk in average when using fuzzy prefix search on the GOV2 collection compared to around 37 MB when exact prefix search was used.

Table 5.10.: Average time required for the intersection of union lists on two-word queries (without caching).

	DBLP	Wikipedia		GOV2	
	non-positional	non-positional	positional	non-positional	positional
ForwardList	9.9 ms	460 ms	-	-	-
Baseline	10.7 ms	270 ms	713 ms	1,922 ms	5,324 ms
Ours	2.0 ms	52 ms	96 ms	272 ms	1,022 ms
Exact prefix search	3.0 ms	41 ms	173 ms	251 ms	1,003 ms

Table 5.11.: Total average fuzzy prefix search query processing times on two-word queries when the index is on disk (without caching).

	DBLP	Wikipedia		GOV2	
	positional	non-positional	positional	non-positional	positional
Baseline	62 ms	1,249 ms	2,206 ms	3,545 ms	8,750 ms
Ours	28 ms	574 ms	652 ms	1,587 ms	3,513 ms
Exact prefix search	16 ms	103 ms	309 ms	706 ms	1,567 ms

Figure 5.4 shows the average query time per keyword length (considering the length of the second keyword). Not surprisingly, the high running times come from the short keywords where the number of hits is larger, but so is the advantage of our algorithm.

Figure 5.5 contrasts the average query processing time for different keyword lengths when the intersection is performed with and without caching (see Section 5.7). Obviously, caching reduces the average running time dramatically since the list intersection is done incrementally, by using previous results.

Table 5.12 shows a break-down of the total query processing time in three categories. The same observation from the previous section applies here too, namely, that the most expensive part of the query processing, which by far exceeds all other costs, is reading large volume of data.

Table 5.12.: Break-down of the total fuzzy prefix search query processing time on Wikipedia.

In-Memory Query Processing	Query Suggestion	Disk I/O
22%	8%	70%

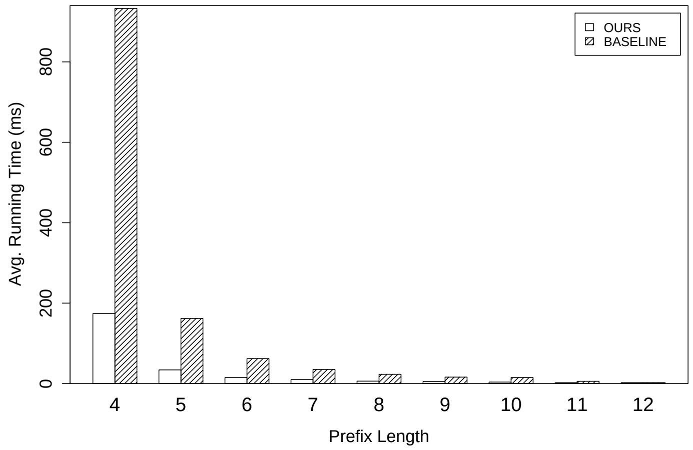


Figure 5.4.: Average query processing times for different prefix lengths (without using caching) on the Wikipedia collection when the index is in memory.

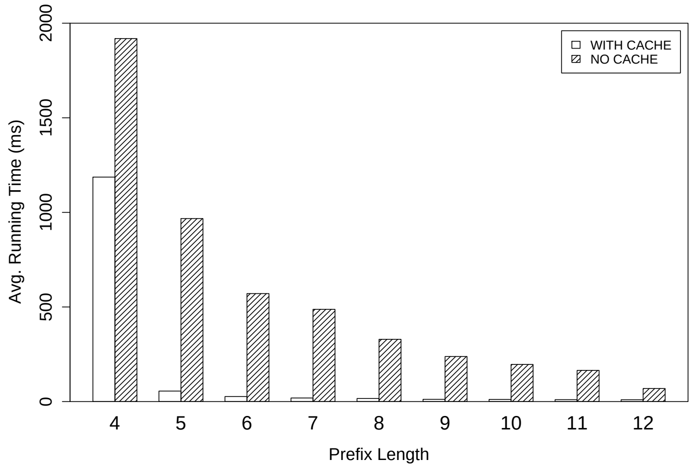


Figure 5.5.: Average query processing times for different prefix lengths with and without using caching on the Wikipedia collection when the (positional) index is on disk.

6. Query Suggestion and Phrase Completion

So far we solved the first (and biggest) part of the problem stated in Definition 2.2.3: given a query Q, find the documents in D that contain the keywords from Q or words similar to them. What remains is to interactively compute a ranked list of query suggestions for Q. In previous work, the query suggestions are typically computed from pre-compiled lists of queries. As a result, no suggestions are shown for queries that are not popular. In contrast, our suggestions are computed based on the indexed collection. More specifically, exactly those queries will be suggested to the user that actually lead to good hits. In addition, we compute phrase completions for the suggestions that are prefixes of popular phrases.

In Section 6.1, we present our query suggestion algorithm. In Section 6.2, we present our phrase completion algorithm.

6.1. Query Suggestion

We start the section by presenting an overview of the related work. In the following Section 6.1.2, we first define a score of a query suggestion and then in Section 6.1.3 propose an algorithm to compute a ranked list of suggestions with respect to this score. Our algorithm takes only a small fraction of the total query processing time. Although relatively straightforward, to achieve the desired speed, careful algorithm engineering is required. In Section 6.1.4, we show how to incrementally compute a ranked list of suggestions when the user types the query letter by letter by using caching.

6.1.1. Related Work

In almost all previous work on this topic, query suggestions come from a pre-compiled list (which itself may be derived from or based on a query log). The focus in these works is not so much efficiency (which is relatively easy to achieve for pre-compiled lists, even with billions of items) but on good similarity measures for finding those queries from the pre-compiled list that match the query typed by the user best. Techniques include vector similarity metrics [Baeza-Yates *et al.*, 2004], query flow graphs and random walks [Mei *et al.*, 2008; Baraglia *et al.*, 2009; Boldi *et al.*, 2009], landing pages [Cucerzan and White, 2007], click-through data [Cao *et al.*, 2008; Song and He, 2010], and cross-lingual information [Gao *et al.*, 2007]. A disadvantage of most of these approaches is that the suggestions are often unrelated to the actual hits.

Bhatia *et al.* [2011] propose a probabilistic mechanism for generating query suggestion from the corpus in the absence of query logs. Their suggestions are common phrases of bounded length from the corpus, and they precompute a special-purpose index (based on n-grams) to provide these suggestions efficiently. Also, they do not deal with misspelled queries. In contrast, our suggestions can be arbitrary queries (as long as they lead to good hit sets), they also work for misspelled queries, and we use the same index as for our fuzzy search.

Kim *et al.* [2011] also do away with query logs. Given a query, their method first retrieves a number of documents for the given query, and from that set constructs alternative queries (that correspond to root-to-leaf paths in a decision tree built over the set) and ranks them using various query-quality predictors. As the authors themselves point out, this can be viewed as a kind of pseudo-relevance feedback. Our approach is similar, except that we use our fuzzy-search index to generate the set of (all matching) documents, that we restrict our query suggestions to those similar to the typed query, and, most importantly, that we consider efficiency which was not an issue in Kim *et al.* [2011].

Google's web search offers query suggestions since 2005. Initially, this service apparently[1] offered exact completions from a pre-compiled list of popular queries. Over the years, the service was significantly extended.

[1] The methods behind Google's suggest functionality have not been published so far. However, given our experience with the service, from using it extensively over the years, and given our experience from our own research on the topic, we consider the claims made in the paragraph above very likely to be true.

In its current state, it offers fuzzy suggestions and it seems that the pre-compiled list does no longer only consist of popular queries, but also frequent or otherwise prominent phrases or combination of such phrases. The basis is still essentially a pre-compiled list, however. Although this approach works surprisingly well for a large class of queries, it also has its obvious limitations. The first limitation is that, unlike in web search, query logs are not always available in domain-specific search environments. The second limitation are expert queries with relatively narrow hit sets: there are simply too many possible ones to include them all in a pre-compiled list. For web search, these are of relatively minor importance, but for vertical search (for example, literature search, intranet search, email search, desktop search, etc.), a large portion of queries are of this type.[2]

The query `beza yates intre` from our example screenshot from Chapter 1 (Figure 1.1) is a good example for this. The intent behind this query is to find papers by Ricardo Baeza-Yates on set/list intersection, and there are, indeed, a number of high-quality hits for the query `baeza-yates intersection` on Google. However, no suggestions are offered for this query, the apparent reason being that it is not in the pre-compiled list of queries. And it cannot be; there are simply too many meaningful queries of such narrowness. In contrast, our approach (applied to a data set containing the mentioned articles) will indeed offer the suggestion `baeza yates intersection`. And the reason simply is that of all the queries similar to `beza yates intre` this is the one with the most or highest-scoring hits.

6.1.2. Score Definition

Given a query $Q = (q_1, \dots, q_l)$, let Q_i be the set of words similar to the ith query word q_i, and let $\mathcal{Q} = Q_1 \times Q_2 \times \dots \times Q_l$ be the set of all candidate suggestions for Q, as defined in Definition 2.2.3. For each suggestion $Q' \in \mathcal{Q}$, we define the following score for Q' with respect to Q:

$$\mathrm{score}(Q', Q) = H(Q') \cdot \mathrm{sim}(Q', Q), \tag{6.1}$$

where $H(Q')$ is a measure for the quality of the set of documents exactly matching Q', and sim measures the similarity between Q' and Q. These two are important features of any reasonable ranking function and formula (6.1) is the simplest meaningful way to combine them.

We now define $H(Q')$. Let $D(Q')$ be the set of documents exactly matching Q', that is, the set of documents that contain each query word q'_i from $Q' = (q'_1, \dots, q'_l)$. For each document $d \in D(Q')$ we define $\mathrm{score}(d, Q')$ as

$$\mathrm{score}(d, Q') = \sum_{i=1}^{l} \mathrm{w}(q'_i, d)$$

i.e., the sum of the weights of all occurrences of a q'_i in d. We take $H(Q')$ to be a weighted sum of all these $\mathrm{score}(d, Q')$:

$$H(Q') = \sum_{d \in D(Q')} \mathrm{W}(d, Q') \cdot \mathrm{score}(d, Q')$$

In the simplest case, the weights $\mathrm{W}(d, Q')$ in this sum are all one. A more sophisticated scoring might assign higher weights to the top-ranked documents in D', the (already computed) set of documents matching Q approximately. Or assign higher weights to those documents in $D(Q')$ where the query words from Q' occur in proximity to each other, and yet higher weights when they occur as a phrase. Our computation described in the next subsection works for any of the above choices of weights.

The second part of our formula 6.1 above, $\mathrm{sim}(Q, Q')$, is supposed to measure the similarity between Q and Q'. A simple definition, based on word Levenshtein distance, would be

$$1 - \frac{1}{l} \cdot \sum_{i=1}^{l} \mathrm{LD}(q_i, q'_i) \tag{6.2}$$

where LD is the word or prefix Levenshtein distance as defined in Chapter 2. Note that this is 1 if $Q = Q'$ and 0 for totally dissimilar Q and Q', for example, when they have no letters in common. For a small number of candidate suggestions, we can also afford to compute a more sophisticated similarity measure by computing a

[2] We say this, in particular, from our experience with running CompleteSearch DBLP for 5 years now, which gets around 5 million hits every month.

(more expensive) generalized Levenshtein distance for each pair (q_i, q'_i), where certain substring replacement operations are counted with a lower or higher cost (see Section 7.2). For example, replacing oo by ue and vice versa might be counted as less than 2 because they sound so similar. We compute the generalized Levenshtein distance for at most 100 candidate suggestions, namely those with the largest $H(Q')$ value.

If we assume that $\text{score}(d, Q') = 1/|D|$ for any document d and any suggestion Q', then $H(Q')$ becomes equal to $|D(Q')|/|D|$, that is, the probability $Pr(Q')$ of seeing Q' in the set of all documents D. In this case, Formula 6.1 can be interpreted as an instance of the *noisy channel model* [Brill and Moore, 2000], where, by using Bayes' theorem,

$$\Pr(Q' \mid Q) \propto \Pr(Q') \cdot \Pr(Q \mid Q') \tag{6.3}$$

In (6.3), the term on the left hand side, $\Pr(Q' \mid Q)$, is the posterior probability that Q' was intended when Q was typed, which can be interpreted as our $\text{score}(Q', Q)$. The first term on the right hand side, $\Pr(Q')$, can be taken as the prior probability of Q', that is, the probability that the user types a query according to a probability distribution $Pr(Q')$. The second term on the right hand side, $\Pr(Q \mid Q')$, is the confusion probability that the user typed Q when intending Q', which can reasonably be assumed to happen with probability proportional to $\text{sim}(Q', Q)$. Note that there are more sophisticated models to estimate $\text{sim}(Q', Q)$ based on query logs; for example see Li *et al.* [2006].

6.1.3. Score Computation

Let Q' be a candidate query suggestion from $\mathbf{Q}$. We now show how to compute $H(Q')$, as defined above, efficiently. For that, we will manipulate with various kinds of inverted lists produced by our query processing (often called result lists in the following). For an arbitrary set or list of documents D, and an arbitrary set or list of words W, let $R(D, W)$ be the inverted list of postings of all occurrences of words from W in documents from D.

In order to compute $H(Q')$, we need the scores of all postings of occurrences of a word from Q' in $D(Q')$, that is, in all documents that contain an exact match for Q'. That is we need exactly the scores from the postings in $R(D(Q'), Q')$. Our query processing as described in the previous subsections does not directly provide us with $R(D(Q'), Q')$. Instead, it provides us with $R(D(\mathbf{Q}), Q_l)$, where $D(\mathbf{Q}) = \bigcup_{Q \in \mathbf{Q}} D(Q)$ is the list of all documents containing a fuzzy match of Q, and Q_l is the set of words similar to the last query word of Q.

Now assume we had the result lists $R(D(\mathbf{Q}), Q_i)$ not only for $i = l$ but for all $i = 1, \ldots, l$. Then, since $D(Q') \subseteq D(\mathbf{Q})$ and $q'_i \in Q_i$, a simple filtering of each of these result lists would give us $R(D(Q'), q'_i)$, the union of which would be exactly $R(D(Q'), Q')$. One straightforward way to obtain $R(D(\mathbf{Q}), Q_i)$ would be to process the query Q with the ith query word swapped to the last position, that is, the query $(q_1, \ldots, q_{i-1}, q_l, q_{i+1}, \ldots, q_{l-1}, q_i)$. However, that would multiply our query time by a factor of l, the number of query words. Another straightforward way to obtain these $R(D(\mathbf{Q}), Q_i)$ would be to compute the intersection of $R(D(\mathbf{Q}), Q_l)$, the result list we actually obtain from our query processing, with $L_{q_i} = R(D(Q_i), Q_i)$, the fuzzy inverted list for the ith query word. However, these L_{q_i} can be very long lists, which we actually avoid to fully materialize for exactly the reason that they can be very long.

We instead consider the result lists of all prefixes of the query Q. Let $\mathbf{Q}_i = Q_1 \times \ldots \times Q_i$ be the set of candidate suggestions for the query $(q_1, \ldots, q_i)$, that is, the query containing only the first i query words from Q. By the iterative left-to-right nature of our query processing, while processing the full query Q, we actually compute result lists for each $(q_1, \ldots, q_i)$, that is, we compute $R(D(\mathbf{Q}_i), Q_i)$, for each $i = 1, \ldots, l$. In the following, we assume that these intermediate result lists are all stored in memory. From these, we compute the desired $R(D(Q'), Q')$'s and $H(Q')$'s simultaneously for each $Q' \in \mathbf{Q}$ as follows

1. Intersect $R(D(\mathbf{Q}), Q_l)$ with $R(D(\mathbf{Q}_i), Q_i)$ to obtain $R(D(\mathbf{Q}), Q_i)$, for $i = 1, \ldots, l-1$. Note that the lists $R(D(\mathbf{Q}_i), Q_i)$ contain the same set of documents with different word occurrences;

2. Simultaneously traverse all the $R(D(\mathbf{Q}), Q_i)$, for $i = 1, \ldots, l$, document by document, in an l-way merge fashion. For each current document d at the frontiers of the lists, collect all postings (containing d). Let Q_i^d be the set of words from Q_i contained in d. Generate the set of suggestions $Q_1^d \times Q_2^d \times \ldots \times Q_l^d$, maintaining their scores $H(Q')$ in a hash-map, and compute the "local" scores $\text{score}(Q', d)$ (if $H(Q')$ should be simply equal to the number of documents that exactly match Q', then set $\text{score}(Q', d) = 1$ and $\text{W}(d, Q') = 1$);

Table 6.1.: Break-down of the average running time to compute the top-10 query suggestions (Wikipedia).

Step 1 (computing $R(D(Q), Q_i)$)	Step 2, 3 (aggregating scores)	Step 4 (sorting)
10%	89%	1%

3. Compute a weight $W(Q', d)$ based on the score of d in D'. If phrases are preferred, then multiply $W(Q', d)$ by a constant larger than one if there are $q'_i, i = 2 \ldots l$ with $pos(q'_i) - pos(q'_{i-1}) = 1$, where $pos(q'_i)$ is the position of q'_i in d. Multiply the computed score(Q', d) by $W(Q', d)$ and update the $H(Q')$'s.

4. After the lists $R(D(Q), Q_i)$ have been traversed, partially sort the candidate suggestions by their aggregated score and output the top k suggestions, for a given value of k.

Running Time

Step 2 and 3 are the most expensive steps of the algorithm. To estimate their running time, let N be the total number of documents in D. Let the length of each fuzzy inverted list L_{q_i} be equal to $N \cdot p_i$, where p_i is the fraction of documents from D in L_{q_i}. The expected length of the result list $R(D(Q), Q_l)$ is then $N \cdot p_1 \cdot \ldots \cdot p_l$. Let n_i be the average number of distinct words in a document with LD within the distance threshold δ from q_i. The average running time of the above algorithm is then dominated by $O(N \cdot p_1 \cdot \ldots \cdot p_l \cdot m_1 \cdot \ldots \cdot m_l)$. For simplicity, assume that each L_{q_i} has the same length, and that each document has equal number of words similar to q_i. The running time is then $O(N \cdot (p \cdot m)^l) = O(|D'| \cdot m^l)$. Since m is usually small, $p \cdot m < 1$. As a result, the running time of the algorithm decreases with the number of keywords l. A reasonable heuristic when computing the lists $R(D(Q), Q_i), i = 1, \ldots, l$ is to include only the posting with maximum score from all posting in a given document that correspong to the i-th keyword in Q, i.e., to include only a single posting per keyword in a document. This means that $m = 1$ and hence the running time reduces to $O(|D'|)$. Table 6.1 shows a break-down of the total running time in 3 different categories by using queries with two keywords.

6.1.4. Cache-based Computation of Query Suggestions

When the user adds a letter to the last query word (without starting a new query word), the computation of the top query suggestions can be done incrementally in time negligible compared to the time to compute query suggestions from scratch. Let $S(Q)$ be the ranked list of triples $(Q'_i, H(Q'_i), \text{sim}(Q, Q'_i))$, where Q'_i is the ith ranked query suggestion for a query Q computed by using the above algorithm. Suppose that Q'' is the newly typed query with $q_l \leq q''_l$, where q_l and q''_l are the last keywords of Q and Q'' respectively. We would like to compute a ranked list of query suggestions for Q'' from the ranked list of query suggestions for Q. Let q'_l be the last keyword of a query suggestion $Q' \in S(Q)$. If $\text{LD}(q''_l, q'_l) > \delta$, then obviously Q' cannot be a suggestion for Q'' and it is removed from the list. Suppose $\text{LD}(q''_l, q'_l) \leq \delta$. Note that $H(Q')$ depends only on the set of documents matching Q' exactly (and hence it is independent of the current query). Since $S(Q'') \subseteq S(Q)$, it suffices to recompute $\text{sim}(Q'', Q'_i)$ for each triple $(Q'_i, H(Q'_i), \text{sim}(Q, Q'_i))$ and sort the new list of triples again with respect to score(Q'_i, Q'').

6.2. Phrase Completion

Automatic completion of popular phrases is a useful feature that can save time and effort. It is supported by all major commercial search engines, at least in certain form. For example, a reasonable completion for `latent semantic` would be `latent semantic analysis` or `latent semantic indexing`. In this section, we show how to compute completions of queries that are prefixes of popular phrases in the absence of query logs.

First, we will provide a short overview of the related work and then present our precomputation algorithm.

6.2.1. Related Work

If query logs are available, phrase completion could be achieved, for example, by traversing a trie constructed over the set of common phrases extracted from the query log. In the absence of query logs, the common

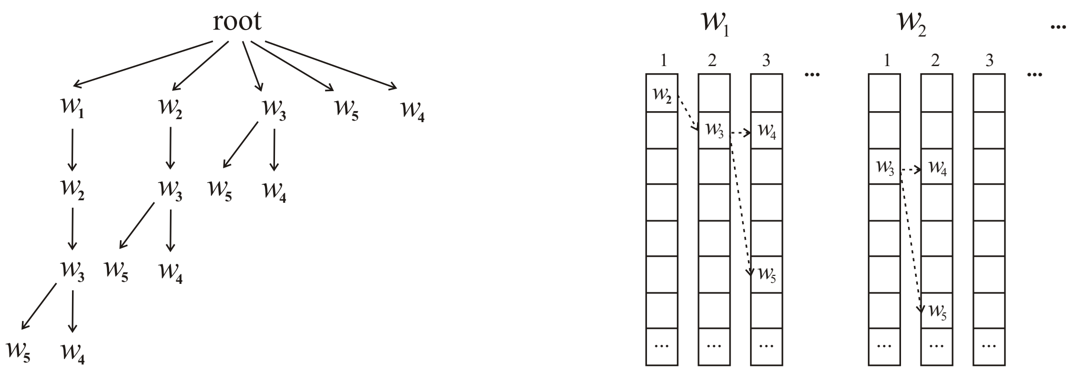


Figure 6.1.: Schematic representation of common phrases with a suffix tree (left) and by using next-word lists (right). The pointers corresponding to dashed lines are not explicitly stored.

phrases must be extracted from the corpus. However, solving this task in general is not easy due to the very large number of phrases in total. One approach is to construct a suffix tree over the corpus. However, since standard suffix trees can grow very large, this solution is not practical on relatively large collections. Nandi and Jagadish [2007] addresses this problem by employing a special type of word-level suffix tree, called *pruned count suffix tree* or PCST. A count is maintained for each node in the tree, representing the number of times a phrase occurs in the corpus. Only nodes with sufficiently high count are retained in order to minimize the size of the tree. Hence, each root-to-leaf path in the tree corresponds to a common phrase. The tree construction goes in two phases. In the first phase, a sliding window is used to scan through the corpus and create an *n-gram frequency table*, for $1 \leq n \leq N$, where N is small. The frequency table contains only frequent n-grams. In the second phase, phrases are added in the tree by using another sliding window and by consulting the frequency table. Only those phrases are stored with frequency potentially above the threshold.

6.2.2. Our Precomputation Algorithm

In what follows, we propose an alternative method to precompute common phrases from the corpus. Given a phrase $w_1, w_2, \ldots, w_i$, we would like to compute common completions from D of the form $w_1, w_2, \ldots, w_n$. Our new method is simple and easy to implement, requires only a single pass over the corpus and has a modest space usage. The basic idea is to precompute M *next-word lists* for each distinct word w, ignoring stop-words, where M is the maximum phrase length. A next-word list of w of order $i \geq 1$ is a list of pairs (w', f), sorted by w', where w' occursm i positions after w and f is the frequency of this event. We would like to store only those pairs (w', f) with f larger than a given frequency threshold. Similarly as before, a sliding window is used to scan through the corpus, ignoring stop-words. At each step, each next-word list of the first word in the sliding window is updated by performing binary search for the corresponding word id. If the word id is not in the current next-word list, then it is inserted at the correct position by shifting all larger elements one position to the right. Whenever the memory limit is reached, each next-word list is pruned by discarding the pairs with frequency equal to 1. A *move-to-front list* is an alternative data structure for stroing word-lists that can make use of the Zipfian distribution of word occurrences. It is searched linearly and the accessed elements are always appended at the front of the list.[3]

To compute a set of common completions of length $i + 1$, observe that w_{i+1} must occur one positions after w_i, two positions after w_{i-1}, etc. Therefore, we intersect the next-word list of w_i of order 1 with the next-word list of w_{i-1} of order 2. The resulting list is then intersected with the next-word list of w_{i-2} of order 3, etc. While performing the intersections, we always take the minimum frequency from the two pairs. Pairs with frequencies below the minimum frequency are discarded. Observe also that the frequency of each pair should be no larger than the frequency of the phrase $w_1, w_2, \ldots, w_i$ itself, which can be estimated in a similar way. Common completions of length $i + 2$ are computed from the common completions of length $i + 1$ in a similar fashion. A heuristic that it turned out to work well in practice is to stop the prediction whenever the estimated

[3] We have not experimented with this approach since the running times of our algorithm on the collections that we tested by using sorted arrays were reasonable.

Table 6.2.: Size and peak memory usage required to precompute common phrases with maximum length of 5 from two of our test collections.

Collection	Collections size	Data structure size	Peak memory usage
DBLP	817 MB	18 MB	116 MB
Wikipedia	21 GB	205 MB	890 MB

frequency of the new phrase drops below a fixed fraction of the estimated frequency of the previous phrase. This is because when we reach the full length of a common phrase, appending another word results in a phrase that is not common anymore and hence its frequency drops sharply.

The compaction in a suffix tree comes from the compact representation of all phrases that share a common prefix. The representation of common phrases with our next-word lists is more compact due to the following. First, unlike a suffix tree, the pointers that lead from one word to the next in the phrase are not explicitly stored. They are, instead, *computed* by performing list intersections (see Figure 6.1). Besides saving space, storing the word-ids in arrays results in much better cache efficiency compared to following pointers to random memory locations. Second, a suffix tree must independently store each suffix of a given phrase. With our approach this is achieved automatically.

In practice, our algorithm requires much less space compared to constructing a suffix tree. For example, the peak memory usage on the English Wikipedia of size 21 GB was 890 MB, while the total size of the next-word lists in uncompressed format with minimum frequency of 10 was 205 MB; see Table 6.2. The running times of our method are well below a millisecond on all test collections. Although the frequencies of the predicted phrases are only estimations, our method works surprisingly well when the frequencies of the phrases are not very low (for example, see Figure 6.2). We compute phrase completion for the query as well as for each of the top suggestions and show a completion in place of the respective suggestion if its estimated frequency is above a threshold. In stricter applications, the most promising suggestions could be refined by launching phrase queries.

Figure 6.2.: Example top-5 common phrase completion for the words “latent“ and “crime“ on the Wikipedia corpus and for the words “suffix“ and “longest“ on the DBLP corpus.

```
latent heat (461)                  suffix trees (132)
latent virus (165)                 suffix arrays (99)
latent image analysis (146)        suffix tree construction (44)
latent semantic analysis (146)     suffix array construction (22)
latent variable (137)              suffix sorting (11)

crime family (8670)                longest common subsequence (137)
crime scene investigation (4602)   longest path (41)
crime drama (2840)                 longest paths (34)
crime fiction (2761)               longest cycles (28)
crime films (1785)                 longest increasing subsequence (22)
```

7. Practically Relevant Extensions

In this chapter, we discuss various relevant extensions to a fuzzy search system that have not been covered in previous chapters. In Section 7.1, we describe how to incorporate a phonetic similarity within our fuzzy search. In Section 7.2, we discuss more powerful versions of edit distance that can model human-made errors, including phonetic errors. In Section 7.3, we talk about advanced ranking and propose a potential top-k query processing approach. In Section 7.4, we propose different approaches to deal with the word-boundary problem. Finally, in Section 7.5, we discuss the practical importance of a search system built with support for different character encodings in mind.

7.1. Phonetic Similarity

Phonetic misspellings arise when the user types a keyword that only sounds like the intended word. Phonetic errors often result in misspellings with Levenshtein distance above a reasonable threshold and thus end up undetected. One way to address this problem is to use *phonetic algorithms*. Given a word w, a phonetic algorithm computes one or more codes that capture the English pronunciation of w. Two words with similar pronunciation should have equal or overlapping set of codes. The oldest and the simplest phonetic algorithm is the *Soundex* algorithm (see The National Archives [2007]) which computes a single code per word. For example, the words `smith` and `smythe` both have Soundex code equal to S540. More sophisticated successors of Soundex include *D-M Soundex*, *Metaphone*, *Double Metaphone* and the *Metaphone 3* algorithm. Phonetic matching can be integrated into our fuzzy search as follows [Manning *et al.*, 2008]:

- Compute the phonetic code of each vocabulary term and build an inverted index from these codes to the original words;
- At query time, do the same with each keyword and search the index to obtain the phonetically similar words. Possible duplicates should be removed if multiple codes per word are generated;
- Add all distinct retrieved words to the list of similar words and process the query as usual.

If efficiency is a concern, we can integrate fuzzy inverted lists of phonetically similar words into our fuzzy index. This can be done by precomputing a fuzzy inverted list for each inverted list in the inverted index. Each fuzzy inverted list comprises all occurrences of the words in the respective inverted list. Since by assumption, all pairs of words with equal codes (should) have similar pronunciation, it follows that the *transitivity property* will be fulfilled. The *no-overlap property* will be fulfilled if each word has only a single phonetic code.

7.2. Generalized Levenshtein Distance

The *generalized Levenshtein distance* allows a more elegant way to tackle phonetic errors. The notion of edit distance can be interpreted as the minimum cost at which one string can be transformed into another given a set of transformations rules. We assume that each transformation is of the type $a \rightarrow b$. In the standard Levenshtein distance, a and b are characters or the empty string and the costs $c(a \rightarrow b)$ are either 0 or 1. In the generalized Levenshtein distance, a and b are arbitrary substring and the costs $c(a \rightarrow b)$ can have arbitrary values, depending on the likelihood of the transformation $a \rightarrow b$. For example, let $c(\text{sch} \rightarrow \text{s}) = 1$ and $c(\text{dt} \rightarrow \text{th}) = 1$. The generalized Levenshtein distance between the strings `schmidt` and `smith` is then 2, while the standard Levenshtein distance is 4. The generalized Levenshtein distance between two words q and w can be computed by using the following recurrence

$$\begin{array}{rcl} C_{0,0} & = & 0 \\ C_{i,j} & = & C_{i,j} + \min\{C_{i-|s_1|,j-|s_2|} + c(s_1 \rightarrow s_2)\} \\ & & \text{for each } 0 \leq i_1 \leq i \text{ and } 0 \leq j_1 \leq j, \\ & & q_{1..i} = q_{1..i_1} s_1,\, w_{1..j} = w_{1..j_1} s_2 \end{array}$$

where $C_{|q|,|w|}$ gives the value of the generalized Levenshtein distance. The algorithm can be obtained by a straightforward adaptation of the dynamic programming algorithm for the Levenshtein distance. Since for each cell (i, j) we need to check all pairs of suffixes s_1 of $q[i]$ and s_2 of $w[j]$ corresponding to a transformation $s_1 \rightarrow s_2$, this algorithm has running time $O(|q|^2 \cdot |w|^2)$. Learning the transformation costs is based on generating and counting the occurrence of each transformation in a given training set of correct word - misspelling pairs. For details, see Ristad and Yianilos [1998] and Brill and Moore [2000].

The generalized Levenshtein distance achieves a higher accuracy compared to the standard Levenshtein distance since it models human made errors better by allowing transformations of richer contextual information. However, it is also considerably more expensive to compute. An algorithm based on traversing tries that computes the generalized Levenshtein distance between a word and the whole dictionary has been proposed in Brill and Moore [2000]. However, no running time comparisons have been made with the standard Levenshtein distance. In general, efficient (indexing) algorithms for generalized Levenshtein distance are a less studied problem since the bulk of the approaches focus on standard Levenshtein distance. Hence, we believe that this field deserves more attention as a future research direction.

7.3. Advanced Ranking and top-k Processing

While the focus in this thesis is mainly on efficiency, proper result ranking is a important aspect in many search applications, especially when the search corpus is large. The ranking that we considered so far is standard, i.e, not different than that of an “ordinary“ search system. Namely, each posting in a given fuzzy inverted lists has a score for the respective document. This score could be, for example, simple TF-IDF or BM25 weighting. Alternatively, we could store the document frequency of the respective word and compute its score on the fly. Given a query Q with l keywords, the score of a document d is the sum of the scores of each keyword q_i in Q. The only difference is that q_i may occur approximately in d and multiple times. Recall that our “extended“ query is given by $Q = Q_1 \times Q_2 \times \ldots \times Q_l$. If multiple words from Q_i occur in d, then we simply compute the sum of the maximum scores:

$$\text{score}(Q, d) = \sum_{i=1}^{l} \max_{w \in Q_i \cap d} \text{score}(w, d)$$

where $\text{score}(Q, d)$ is the score of document d and $\text{score}(w, d)$ is the score of a word w in a document d. A relevant issue that we have not considered is how to incorporate the Levenshtein distance of a matching word in its score. For example, if a keyword matches only few documents exactly, should they have higher ranks than the many documents matched approximately? In certain scenarios, such as product search where the data is clean, this might make sense. However, if our keyword is a misspelling and our not-so-clean corpus by chance has few documents containing it exactly, it might not be so useful to see these documents on top. The simplest formula reconciling both scenarios is given by

$$(1 - \alpha) \cdot \text{score}(w, d) + \alpha \cdot \left(1 - \frac{\text{LD}(q, w)}{\max(|q|, |w|)}\right) \cdot \beta$$

where $0 \leq \alpha \leq 1$ and $\beta > 1$. For example, setting $\alpha = 1$ removes the influence of the scores, while setting $\alpha = 0$ removes the influence of the (normalized) Levenshtein distance. A different formula that permits giving large preference to documents containing the keywords exactly, while documents with equal (or similar) Levenshtein distance are ranked by their scores, is given by

$$\left(1 - \frac{\text{LD}(q, w)}{\max(|q|, |w|)}\right)^{\gamma} \cdot \text{score}(w, d)$$

where the parameter γ sets the influence of the Levenshtein distance.

Another important issue is which hits to show to the user in the first place. In our original problem definition we show mixed hits for all matching l-tuples of words in Q. Which documents are presented to the user then depends merely on the ranking. Another approach is to show hits only for the few top query suggestions. Incorporating the Levenshtein distance into the scores in this case is not required. However, one can also argue that with proper ranking, showing mixed hits could provide potentially useful diversity to the user.

Top-k query processing is relevant extension to fuzzy search when searching in large text collections. This is because, first, ranked retrieval makes more sense for large collections, and second, fully materializing the lists of all matching postings in this scenario can be quite expensive. Top-k query processing requires storing the fuzzy inverted lists sorted by score. Recall that we need several fuzzy inverted lists for each keyword. Obviously, it is completely impractical to read each of these lists from disk and then sort (merge) them in memory. Instead, we employ a priority queue on top of the lists similarly as in Wang *et al.* [2009] and allocate a buffer to each list. The buffer is refilled every time it gets empty. Whenever required, we access and pop the top element in the priority queue as in multi-way merge. This avoids reading the entire list from disk. We could then use threshold-based algorithms to compute the exact top-k results, such as Fagin *et al.* [2001]; Bast *et al.* [2006], or pruning strategies that compute approximate top-k results, such as Moffat and Zobel [1996a]; Theobald *et al.* [2004]; Lester *et al.* [2005]. We have not experimented extensively with these techniques and their effectiveness in our setting in practice remains open.

7.4. The Word-Boundary Problem

Another practically important issue is that certain phrases or compound words are sometimes written as one word, and sometimes as multiple words (e.g. `memorystick` vs. `memory stick`). In some literature this is known as the word-boundary problem [Kukich, 1992]. It seems reasonable to simply not distinguish between such variants. One way to achieve this is to index each phrase under both, the one-word and the separate-word variants, provided that we have a list of known phrases.

A different approach is to compute and suggest the alternative variant of the query whenever it leads to better or to more hits. To accomplish this, we have to compute the optimal sequence of constituent words for a phrase that is incorrectly typed as a single word. By “optimal“ here we mean the most common phrase or the phrase with the highest likelihood. In practice, this can be done whenever the number of hits is below a threshold; whenever the other variant results in significantly larger number of hits or simply whenever the given query is very rare in our query log.

If we assume absence of spelling errors, then both problem can be easily solved by matching the concatenated version of the phrase (with all space characters being removed) in the concatenated version of the query log in which pointers are kept to the original phrases. If there are multiple matches, we suggest the match with the highest frequency. For example, `back bone` would then match `backbone` and `washingmachine` would match `washing machine`.

Making this feature error-tolerant is more challenging. For example, it is reasonable to demand that a fuzzy search for `memoristick` matches an occurrence of `memory stik`, but that requires either figuring out the position of the missing space in the query or figuring out and indexing all (common) spelling variants of the common phrase `memory stick`. Solving either of these two tasks efficiently is not easy, especially for phrases with more than two words. If query logs are available, the alternative approach is to fuzzy match the concatenated version of the query in the concatenated version of the query log. For example, `memoristick` would then match `memory stick` and `disch washer` would match `dishwasher`.

7.5. Support for Different Character Encodings

All of our algorithms have been implemented in C++ with support for different character encodings. Our code makes extensive use of templating to support single byte, such as ISO 8859-1, as well wide-byte or multi-byte character encodings, such as UTF-8. Although this is quite a hassle implementation-wise, it is of great practical importance. For example, any approximate matching algorithm that has been implemented by using a string data structure based on bytes, such as `string` or `char*`, would fail on texts encoded in unicode, where each characters consists of up to 4 bytes. Moreover, UTF-8 has become the dominant encoding for the Word-Wide-

Web, accounting for more than half of all web pages.[1]

[1] http://googleblog.blogspot.de/2010/01/unicode-nearing-50-of-web.html

8. Index Construction

In this chapter, we describe how to construct our fuzzy indexes efficiently. Our fuzzy search in practice is realized via the prefix search and autocompletion mechanism of the *HYB index* [Bast and Weber, 2006]. The HYB index is a data structure that supports variety of other advanced queries via the same simple yet versatile mechanism. The one question that was left open in Bast and Weber [2006] was how to construct the HYB index efficiently. The construction algorithm in this work constructs and then post-processes an ordinary inverted index (INV), yielding a total index construction for HYB that is about *twice as much* as that of INV. In this chapter, we show that HYB can be constructed at least as fast as INV, and often up to twice as fast. This is remarkable in two respects. First, because HYB is as practical but more powerful than INV with regard to efficient support of advanced queries. Second, because fast index construction for INV has been the subject of extensive research and is well understood, leaving little room for further improvement of the state of the art.

We first describe how to efficiently construct a HYB index for exact prefix search. This will require only a single pass over the corpus. Then we show how to modify the algorithm to construct a HYB index for fuzzy search at the price of an additional pass. The second pass is necessary since the dictionary of the corpus is required in advance.

In Section 8.1, we start the chapter by providing the necessary background of INV and HYB, including previous work. In Section 8.2, we provide an overview of our construction algorithm and point out challenges and efficiency issues that have not been addressed in previous work. In the following Sections 8.3 and 8.4, we describe the in-memory part of our algorithm. In Section 8.5, we describe the on-disk part of our algorithm. In Section 8.6, we describe the modifications required to construct a fuzzy search index. In Section 8.7, we present the experimental results.

8.1. Background and Related Work

8.1.1. The Inverted Index

An inverted index maintains the set of distinct words of a text collection in a *vocabulary*, with a *posting list* or *inverted list* assigned to each word. A posting list comprises the list of postings for that word and it is usually stored contiguously on disk. Each posting corresponds to a word occurrence and is of the form $(d, f_{d,w})$ or $(d, s_{d,w})$, where $f_{d,w}$ and $s_{d,w}$ represent the frequency and score of the word w in the document d, respectively. A word-level or positional index additionally includes the word positions in the document. Figure 8.1 gives an example of an inverted list.

An extensive amount of research has been done on static inverted index construction and many inversion approaches have been proposed, although only few are scalable in practice [Witten *et al.*, 1999]. We compare ourselves against the state-of-the-art inverted index construction proposed in Heinz and Zobel [2003b] (referred to as the *single-pass approach*) which improves the well known *sort-based* approach which is considered as one of the most efficient approaches described in the literature [Heinz and Zobel, 2003b; Witten *et al.*, 1999].

Both approaches are *in-memory*, as the inversion is done in main memory, and *single-pass*, as only one parsing pass over the uncompressed data is required (note that our definition of single-pass is stricter since we allow only a small number of word occurrences to be touched twice). Two-pass approaches on the other hand are slow but memory efficient since the number of postings per indexed word is known. Hence the sizes of all in-memory and on-disk vectors can be easily calculated and effective compression schemes can be applied [Heinz and Zobel, 2003b; Witten *et al.*, 1999]. Disk-based approaches can invert collections of any size but suffer from excessive cost since traversal of on-disk linked lists of non-adjacent postings is required [Harman and Candela, 1990; Rogers *et al.*, 1995]. In-memory approaches, on the other hand, maintain an in-memory data structure for posting accumulation, either a linked list or a dynamically growing array. However, since text collections are typically much larger than the available main memory, the index is split into smaller runs each of which is in-memory inverted and written out to disk. The final index is obtained by merging the on-disk

Table 8.1.: Positional inverted index. A posting list consists of a word (term) and postings. Each postings is a triple consisting of a doc-id (first row), a position (second row) and a score.

	D9002	D9002	D9002	D9004	D9004
algorithm	5	9	12	4	21
	0.8	0.8	0.8	0.2	0.2

runs through a multi-way merge [Heinz and Zobel, 2003b; Moffat and Bell, 1995]. An extensive amount of collected work on inversion approaches can be found in Witten *et al.* [1999].

The sort-based approach consists of the following basic steps: (*i*) a word to word-id mapping is maintained through hashing and the available memory is filled with postings that come from the incoming parsing stream; (*ii*) when the main memory limit is reached, the postings are sorted, compressed and written to disk as a new run; (*iii*) after the whole collection has been processed a multi-way merge is performed to obtain the final index. The merge can be performed either *in-situ*, for additional index permuting cost [Moffat and Bell, 1995; Witten *et al.*, 1999], or with a temporary file using roughly twice as much space as the size of the index.

The single-pass approach from Heinz and Zobel [2003b] modifies steps (*i*) and (*ii*) as follows. First, instead of sorting postings, a dynamic array that accumulates the postings from the posting stream in compressed format is assigned to each index word; and second, words instead of word-ids are included in the runs and thus no word to word-id mapping is required. The advantage of the modified version is that sorting of a large amount of in-memory postings is avoided and that the vocabulary can be flushed to disk and the memory freed whenever a new run is written out to disk. The reported improvements range from 15% up to 20%.

In addition, Heinz and Zobel [2003b] propose an interesting idea to reduce the merge cost that is similar to our construction algorithm: the index is partitioned into b buckets so that each bucket is responsible for a non-overlapping range of lexicographically adjacent index words. The ranges of words are determined by building an in-memory inverted index for a snapshot of documents from the collections and then splitting the accumulated vocabulary into ranges so that each range corresponds to a set of posting lists of roughly the same memory size. Once the memory is exhausted a block is flushed to disk as a run. Hence each bucket is an index on its own small enough to fit in memory so that it can be independently in-memory merged. The improvement in the running time is, however, only marginal. The reason for this is that the improvement aims at reducing the number of disk seeks during the merging, which not the bottleneck in the state of the art inverted index construction (see Section 8.7.3).

8.1.2. The HYB Index

The HYB index does not maintain an explicit word-level vocabulary but rather a vocabulary of word ranges. The number of word ranges is typically a few thousand times less than the number of distinct words. Each word range corresponds to a *block* of postings (HYB block) that is analogous to an inverted list. The word ranges are defined by a sequence of *block boundary words* $w_0, \ldots, w_k$ such that block i contains all words in the range $(w_{i-1}, w_i]$, where w_0 is some word lexicographically smaller than all words in the collection. A posting in a HYB block is a document-id, word-id, position, score quadruple. In each HYB block, the postings are sorted by document-id and position (not by word-id). Both documents and words have contiguous ids. Word-ids are assigned to words in lexicographical order; this is key for the fast processing of prefix queries with HYB. For an example of a block that corresponds to the word range (alg, alz] see Figure 8.1. In this example, the first list entry says that the word "algorithm" occurs in a document with ID D9002 at position 5 with a score of 0.8.

One of the key results from Bast and Weber [2006] is that if these blocks are of roughly equal volume $\varepsilon \cdot N$, where N is the number of documents and ε is some constant, then HYB can be stored in space $1 + \varepsilon$ times that of INV.

The HYB index allows efficient computation of *autocompletion* queries. Formally, an autocompletion query is a pair (D, W), where W is a range of words (all possible completions of the last query word) and D is a set of documents (the hits of the preceding part of the query). A result of an autocompletion query is a subset $W' \subseteq W$ of words that occur in D, as well the subset $D' \subseteq D$ of their matching documents. Both D' and W' should be computed in ranked order. Less formally, imagine a user of a search engine typing a query. Then with every letter being typed, in time less than it takes to type a single letter, we would like a display of completions of the

Figure 8.1.: Positional HYB index. A HYB block consists of a word range formed by two consecutive block boundaries. Each postings is a quadruple consisting of a doc-id (first row), a word-id (second row), a position-id (third row) and a score.

[alg, alz)	D9002	D9002	D9002	D9003	D9004
	algorithm	algorithmica	algorithmic	algae	algorithm
	5	9	12	54	4
	0.8	0.8	0.8	0.9	0.2

last query word that would lead to good hits. At the same time the best hits for any of these completions should be displayed.

Processing an autocompletion query with the HYB index goes in three simple steps as follows.

1. Find each block B that correspond to a (precomputed) word range that overlaps with the last part of the query (it is not hard to prove that there is only one such block in average);

2. Intersect each B with D to compute W'_B and D'_B;

3. Compute D' by a multi-way merge and compute W' by a simple linear-time sort into W buckets.

Another key result from Bast and Weber [2006] is that if the blocks are chosen of sufficiently large volume, then the average autocompletion query processing time is linear in the number of documents containing it. For proofs and more details (e.g., document and word ranking) the reader should refer there.

8.2. Algorithm Overview

To be able to construct a HYB index we must know the block boundaries in advance so that at parsing time we can determine the block to which a given word-id belongs. This could be trivially done by a full pass over the data, counting the frequency of each word and then computing the prefix sums. This is indeed necessary to achieve fuzzy search. However, in case of exact search, inspired by parallel sorting algorithms, we show that is possible to compute good estimates of the block boundaries by sampling only a logarithmic number of random passages in the given document collection (Section 8.3).

Given the sequence of block boundaries, a straightforward approach to building the HYB index with a single pass would be as follows. Maintain a dynamically growing in-memory data structure of postings for each block, and for each word parsed, append the corresponding posting to the array of the block to which it belongs. When all words have been parsed, compress the blocks one after the other, and write them to disk. As is the case with the inverted index construction, the first obvious problem is that the size of the in-memory data structures will at some point exceed the total available memory. An obvious solution is to process the collection in parts by imposing a limit on their in-memory size. Once the collection is fully scanned, the fragmented parts (partial blocks) should be merged to produce the final index. This simple approach has the following efficiency issues.

The first efficiency issue is index merging since a full additional pass over the compressed data is required. We show that unlike INV construction, for the HYB construction we can avoid index merging by appending the in-memory blocks to their corresponding positions on disk in-place. This process is explained in more detail in Section 8.5.

A second efficiency issue not considered in the previous work of Heinz and Zobel [2003b] is the cache efficiency of the posting accumulation. Namely, even though well approximated by Zipfian distribution and hence with a good locality of reference, we show that a significant fraction of the postings impose cache misses when appended to their inverted lists (HYB blocks). This is due to the fact that the number of inverted lists is typically much larger than the number of lines of the L1 data cache. In Section 8.4.1 we propose a multi-level posting accumulation scheme with better locality of access that can significantly improve the in-memory inversion performance for both, HYB and INV.

A third efficiency issue is the compression of word-ids. Unlike INV, HYB requires storing the word-ids as a part of each posting. However, the word-ids cannot be gap-encoded as they come in random order and have to be entropy-encoded instead. Near entropy-optimal compression could be, for example, achieved by arithmetic

Table 8.2.: Standard deviation of the block sizes computed by a full pass given as a percentage of the ideal block size n/k, where $k = 2000$ for all three collections.

	DBLP	Wikipedia	TREC GOV2
Standard deviation (%)	12.7%	1.6%	6.4%

encoding [Witten *et al.*, 1999]. In Section 8.4.2 we propose a much faster two-pass compression scheme that yields only a slight loss in the compression ratio. Furthermore, storing word-ids requires a permanent in-memory word to word-id mapping. Section 8.4 makes the simplifying assumption that the vocabulary fits in main memory. In Section 8.7.5 we propose a refinement of our basic algorithm that addresses this issue.

In Section 8.7 we experimentally compare our HYB index construction algorithm against the state-of-the art inverted index construction algorithm from Heinz and Zobel [2003b]. As a reference we also compare ourselves to the very fast and well engineered Zettair system.

8.3. Computing Block Boundaries

We already mentioned that the block boundaries of the HYB index can be easily computed by scanning the whole collection. Having the word frequencies, the boundaries are the words from the vocabulary that split the collection into unions of posting lists with roughly the same number of occurrences. In this section we show how to compute the block boundaries by only a logarithmic number of accesses to the given collection. The basic idea is related to the idea of splitter selection in the parallel sorting literature (Samplesort, [Grama *et al.*, 2003]). Let n be the collection size in total number of occurrences and let k be the number of HYB blocks. The sampling lemma below shows how to compute block boundaries from a sample of word occurrences so that the resulting blocks are of size less than $a \cdot n/k$ with high probability, where $a > 1$ is an arbitrary constant and n/k is the ideal block size.

8.3.1. Sampling Lemma

Lemma 8.3.1. *Pick $s \cdot k$ numbers from the range $1..n$ uniformly at random and independently from each other. Sort these numbers and consider the k integers $x_1, ..., x_k$ whose rank in the sorted sequence is a multiple of s. Let $B_1, ..., B_k$ be the block sizes induced by splitting the range $1...n$ according to $x_1, ..., x_k$. Let $B_{max} = \max\{B_1, ..., B_k\}$. Then*

$$Pr(B_{max} > a \cdot n/k) \leq n \cdot exp(-s \cdot K) \tag{8.1}$$

where $K \approx a - ln(a) - 1$.

Proof. For brevity the proof is given in the appendix. We note that there is an alternative proof of the lemma based on Chernoff bounds with a looser upper bound that for the same failure probability requires close to twice as many sampled words. □

Example 8.3.2. *Let $k = 2000$, $a = 1.5$, $n = 10^{10}$ (a dataset with 10 billion occurrences) and a failure probability of 10^{-10}. Then a sufficiently large s so that $Pr(B_{max} < 1.5 \cdot n/k) \leq 10^{-10}$ is 512, which means that a sample of 0.01% of the full collection is enough. Moreover, increasing the dataset by 10 times requires increasing s by roughly 30 (or 6% which is 60,000 more occurrences in total)*

We note that for the above calculations and discussion, we assume that the list of all occurrences (considered sorted by word) can be split into blocks at arbitrary positions. However, in practice we do not put the word boundaries in the middle of a run of occurrences of the same word. This gives blocks of size at most those predicted by the lemma, except for very frequent words, which get a block on their own, and which in principle can become arbitrarily large, depending on the frequency of the word(s). Nevertheless, it is desirable in practice that very frequent prefixes such as `pro`, `com`, `the`, etc. get blocks on their own. Table 8.2 shows the standard deviation of these real block sizes from the ideal n/k.

We also note that although sampling causes some word occurrences to be visited twice, it does not violate our definition of single-pass since the number of drawn samples is very small compared to the total number of postings in the collection (about 2% in all our experiments).

8.3.2. Query Time

The following lemma says that the blocks obtained via sampling give a query time on the same order of magnitude as we would obtain it for the ideal blocks.

Lemma 8.3.3. *Assume block boundaries are computed as in Lemma 8.3.1. Then, for an arbitrary given query, the query time is $O(q \cdot n/k)$ with high probability, where q is the number of blocks that contain a word / prefix from the query.*

Proof. By Lemma 8.3.1 the maximal size of a block is $O(n/k)$ with high probability. The lemma then follows from the observation that each query word requires a scan of each block containing that query word. □

In practice, blocks are quite large (there are only a few thousand blocks in all our experiments). Therefore, typically only one block needs to be processed per query word and q is just the number of query words.

Also note that the above lemma holds for block boundaries that split the blocks at arbitrary positions. Again, for the actual block boundaries, a query word that lies in a block containing a very frequent word will require larger processing time, depending on the size of that block. In practice, we either accept these larger processing times, or we remove very frequent words (stop words) from the index.

8.3.3. IO-Efficient Sampling

Lemma 8.3.1 asks for a random sample of postings with each posting drawn uniformly and independently of other postings. The straightforward algorithm for drawing such a sample would require one random disk seek per posting. This can be prohibitively expensive if the sample size is large: to pick a sample of 0.01% of all postings in the example above would take around 10,000 secs, in that time we could scan 500 GB of data. We therefore use a variant of random sampling, namely *cluster sampling* [Cochran, 1977], where for each random access we read a whole passage of text from disk and draw a random sample of postings from this passage. Let P be the number of sample passages (number of affordable disk seeks) and p be the probability of drawing a certain occurrence within a passage. The passage size at each access then must be equal to $s \cdot k/(P \cdot p)$ occurrences. Table 8.3 shows the running time of random and area sampling for different sample sizes. Obviously, the main cost in area sampling comes from the number of passages and not from the size of the sample.

The assumption behind cluster sampling is that the passages have approximately equal distributions. As this is not always the case in practice, the price paid is a larger sampling error [Cochran, 1977]. The sampling error in our case depends on P and p. A good trade-off between the sampling error and the sampling time was achieved for $P = 5000$ and $p = 0.1$. Smaller values of p did not seem to involve a significantly better approximation of the block boundaries. Table 8.4 shows that computing the block boundaries using cluster sampling with these values of the parameters results in block sizes reasonably close to those computed by a full pass. We measure the deviation in two ways: (i) by the standard deviation of all block sizes from the ideal block size; and (ii) by Kullback-Leibler divergence, considering the block sizes obtained by both sampling methods as histograms. The intuition here is that if area sampling is biased and significantly deviates from uniform random sampling, then the resulting “distribution“ of block sizes computed by sampling will significantly deviate from the “distribution“ of block sizes computed by a full pass. We observed essentially the same results on all of our test collections.

8.4. In-memory HYB Inversion

The posting accumulation from Heinz and Zobel [2003b] works by assigning a single dynamic array to each inverted list. The postings coming from the parser are then stored by using on-the-fly compression. Dynamic arrays (also dynamic bit-vectors or doubling vectors) are in fact the standard, as well as the optimal, choice for dynamically growing data structures for index construction [Zobel and Moffat, 2006]. Note that any array data structure could serve as a bit vector, for example `vector<int>` in C++/STL. Buttcher and Clarke [2005] observed that by maintaining the array's dynamic growth with a special function, almost as 75% additional space can be saved as when the array size is known in advance.

Table 8.3.: Running time of random disk sampling compared to area disk sampling for different sample sizes and number of passages (accesses). The ratio between the passage size and the number of samples drawn is fixed to 0.1.

# samples	# passages	Area sampling time	Random sampling time
10^4	1000	4.3 sec.	42.1 sec.
	5000	19.7 sec.	
	10000	40.1 sec.	
10^5	1000	4.6 sec.	302.9 sec.
	5000	21.5 sec.	
	10000	40.4 sec.	
10^6	1000	7.1 sec.	886.7 sec.
	5000	27.1 sec.	
	10000	47.5 sec.	

Table 8.4.: Standard deviation (given as percentage of the mean block size) calculated over all HYB block sizes when computed with sampling (random and area, defined in Section 8.3.3) and with a full pass. The last column shows the Kullback-Leibler divergence when the block size are considered as probability distributions (see last paragraph of Section 8.3.3). All numbers are averages of 100 experiments carried out on 1 GB dataset with 2000 word boundaries (HYB blocks) .

Sampling	Sample Size	Standard deviation (%)	Kullback-Leibler div.
Full dataset	100%	12.7%	0
Random	0.06%	15.1%	1.52
	0.6%	13.0%	1.45
	1.2%	12.8%	1.37
Area	0.06%	15.2%	1.43
	0.6%	12.9%	1.45
	1.2%	12.7%	1.46

8.4.1. Multi-level Posting Accumulation

A weakness of the above scheme is its cache inefficiency since the number of inverted lists is usually in the order of millions while the number of L1 data cache lines is in the order of hundreds. Even though the distribution of postings is Zipfian and exhibits a good locality of access, we experimentally show that a significant fraction of the postings will impose cache misses when appended to their lists. As we will see later, this badly hurts the efficiency of the in-memory part of the index construction. Interestingly, to the best of our knowledge, this issue has not been studied in previous work. Note that since the number of HYB blocks is much smaller than the number of inverted lists, the cache inefficiency of HYB posting accumulation is less severe.

We propose a multi-level posting accumulation scheme by grouping consecutive HYB blocks in groups so that the total number of groups is close to the number of L1 data cache lines. The postings from each group are, in a tree-like fashion, assigned to the next level of groups until each group is comprised of a single block. The idea is to assign each group to only a small number of groups at the next level of groups so that at each assignment of postings there are virtually no cache misses. If l levels of assignment are required to achieve this, then the number of groups per level should be $\sqrt[l]{k}$. To illustrate this by example, say we have 2000 blocks and 50 cache-lines. Then $l = 2$ is sufficient since $\sqrt{2000} < 50$. At the first level of assignment a posting that belongs to block i will be assigned to group $\lfloor i/\sqrt{2000}\rfloor$. At the second level of assignment, the postings coming from group j will be assigned to the blocks ranging from $j \cdot \lfloor\sqrt{2000}\rfloor$ to $(j+1)\cdot\lfloor\sqrt{2000}\rfloor$. Figure 8.2 gives an overview of this process. The price paid is moving each posting more than once. Note, however, that a cache hit can be several times faster than a cache miss.

In the following we give a theoretical evidence that under certain assumptions two-level posting accumulation is significantly faster than one-level posting accumulation.

Lemma 8.4.1. *Consider a fully associative cache of size c and HYB blocks of equal size. Let a cache miss take m times longer than a cache hit. Two-level posting accumulation is then faster by a factor of $m/2 - g$, where g is usually a small function that depends on k.*

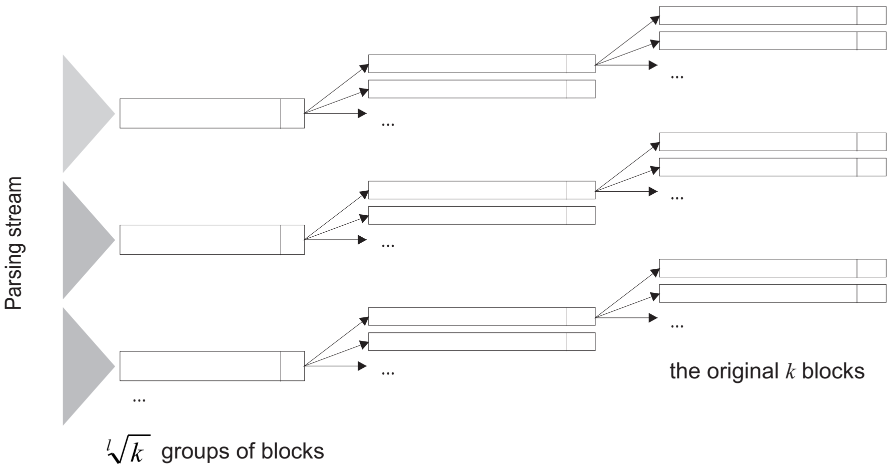


Figure 8.2.: Multi-level posting accumulation.

Table 8.5.: Throughput given in MiB/s for one-level posting accumulation, two-level posting accumulation and sorting-based inversion on the inverted index (INV) for different values of the α parameter of the Zipfian distribution of word occurrences.

$\alpha = 1.0$		$\alpha = 1.1$		$\alpha = 1.2$	
sorting:	50 MiB/s	*sorting*:	60 MiB/s	*sorting*:	73 MiB/s
$l = 1$:	46 MiB/s	$l = 1$:	84 MiB/s	$l = 1$:	150 MiB/s
$l = 2$:	221 MiB/s	$l = 2$:	295 MiB/s	$l = 2$:	314 MiB/s

Proof. The relatively simple proof is based on calculation of the total number of cache misses according to the assumptions and it is given in the appendix. □

Example 8.4.2. *Consider a 8 KB cache with 64 B cache lines. Assume that an L1 cache hit requires 2 cycles while 8 cycles are required for a cache miss (which would still be an L2 cache hit due to the 2-level cache hierarchy). Then according to the above model, the two-level posting accumulation is roughly 2 times faster when* $1,000 \leq k \leq 10,000$.

Indeed, the best result in practice was achieved for two-level posting accumulation. The left side of Figure 8.3 shows the number of cache-misses of one and two-level HYB posting accumulation on a 2-way associative, 64 KB L1 data cache with 64B cache lines, computed using the `valgrind` tool[1]. The right side of the figure shows a comparison of the efficiency of one and two level posting accumulation for both, INV and HYB. The efficiency of one-level HYB posting accumulation is highly affected by the number of blocks and in fact can approach that of INV for a large number of blocks and small cache sizes. The efficiency of the two-level posting accumulation, on the other hand, is only slightly affected.

The number of cache misses in the posting accumulation for INV depends on both, the number of inverted lists and the value of the α parameter of the Zipfian distribution of word occurrences. According to Breslau *et al.* [1999] the asymptotic cache hit ratio of a cache with c cache lines is given by $H_{c,\alpha}/H_{n,\alpha} \approx (c^{1-\alpha} - 1)/(n^{1-\alpha} - 1)$ where $H_{n,\alpha} = \sum_{i=1}^{n} 1/i^{\alpha}$ is the generalized harmonic number of order n of α and n is the number of inverted lists. However, after grouping the inverted lists in groups, the distribution of occurrences in the groups is not Zipfian anymore and thus the total cost is hard to approximate. Still, the results given in Table 8.5 show that the improvement on INV is in fact more dramatic than that of HYB when two-level posting accumulation is used and varies from a factor of 2 for $\alpha = 1.2$ to a factor of 5 for $\alpha = 1.0$. One caveat here is that two-level posting accumulation is not easy to apply on INV since all distinct words are not known in advance. Another caveat is that the actual improvement depends on the amount of processor cache available.

[1] `http://valgrind.org`

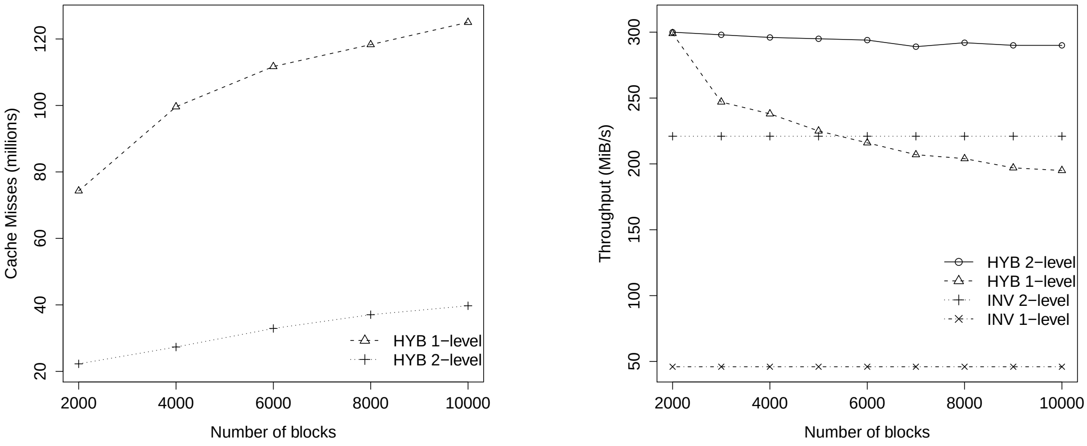


Figure 8.3.: Left: Total number of cache misses for 1 and 2-level HYB posting accumulation for total of 300,000,000 accesses (64K, 2-way associative cache, with 64B cache lines). Right: Throughput (given in MiB/s) of HYB and INV posting accumulation approaches used to in-memory invert 100 million occurrences for $\alpha = 1.0$ (defined in Section 8.4.1). Each posting is assumed to have 4 bytes.

8.4.2. Fast Compression of HYB Blocks

Consider the following example of a HYB block that corresponds to the word range [5,133):

3	3	3	5	5	5	5	5	6	6	6	6	8	8	8
1	4	5	2	5	9	11	15	7	8	12	15	7	12	15
7	122	5	113	7	24	7	122	88	93	7	113	122	5	93

The first row corresponds to doc-ids, the second to positions and the third to word-ids (each word in the block is replaced by its word-id). One of the differences to INV is that the word-ids are part of the postings and have to be compressed just like the doc-ids and the positions. The latter means that a fast compression of word-ids is required. But unlike the doc-ids and the positions, which come in ascending order and could be compressed by gap and universal encoding (just as done for INV), the word-ids come in random order and have to be entropy encoded. One problem with entropy-optimal encoding like Huffman or arithmetic encoding is their speed. Gap combined with universal encoding, on the other hand, is a fast coding scheme which involves simple word-level arithmetic operations [Moffat and Zobel, 1996b].

The compression scheme that we propose is based on the assumption that the word-ids have near Zipfian distribution. Given this, it is not hard to show that universal encoding that assigns $\sim \log x$ bits to a word of rank x obtained by sorting the word-ids in order of descending frequency, is near entropy optimal as well. To see this simply observe that if there are k distinct words in a block and if the relative frequency of the i-th most frequent word is Ω/i, then the per-item entropy is equal to

$$\sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 \frac{i}{\Omega}$$

while the average per-item-space consumption would be equal to

$$c \cdot \sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 i \tag{8.2}$$

for some (small) constant factor $c > 1$. We refer to this scheme as *Zipf compression*. The scheme requires three passes: a pass to obtain the frequency counts (linear in the number of word occurrences), a pass to sort the distinct word-ids with respect to their frequency counts (super-linear in the number of distinct word-ids) and a third pass where each word-id is replaced by its rank. Assume that instead of sorting, the rank of each word is approximated by a move-to-front (MTF) transform over the input which can be computed more efficiently in a single linear scan of the list. In a move-to-front transform, the next element is always appended at the beginning of a list with the intuition that frequent elements are likely to end up being not far from the front of the list and thus obtain smaller ranks. The following lemma shows that this gives us an efficient compression algorithm at the price of a small loss in the compression ratio.

Lemma 8.4.3. *Given a Zipfian distribution of the input with independent consecutive elements, the ranks obtained by a MTF-transform have expected values that are not much larger than the true ranks (in particular, the ranks are less than a factor of* $1/\Omega$ *away from the true ranks, i.e., more skewed distribution results in ranks that are closer to the true ranks).*

Proof. Let X_r be a random variable defined as the rank of a word w obtained by a MTF-transform over the input. Observe that the value of X_r is equal to the number of words in front of the first occurrence of w, ignoring duplicates (counting from the end of the original input list). Let j be the true rank of w (obtained by sorting) and let X_j be a random variable defined as the number of words (including duplicate words) in front of the first occurrence of w. Let $p = \Omega/j$ be the probability of observing w, where $\Omega < 1$ is a normalization constant that depends on the distribution. Observe that X_j has a geometric distribution with expectation $E[X_j] = 1/p = j/\Omega$. This gives us an upper bound on the expected value of X_r as $X_r \leq X_j$. □

According to the above lemma, the average per-item-space consumption is now less than

$$c \cdot \sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 \frac{i}{\Omega}$$

which is only slightly larger than the value given in Equation 8.2 (note that $\Omega < 1$). The loss in compression ratio due to the MTF transform on our test collections in practice was not far from what is predicted by our model: less than 1% more total space required when indexing Wikipedia and about 1% more total space required when indexing TREC GOV2 compared to when the ranks were obtain with sorting the word-ids. Regarding the speed, our word-id compression scheme is twice as fast as coding word positions with gaps and Elias gamma code (see Table 8.6).

The above lemma does not ensure that the few most frequent word-ids that make most of the input get their true ranks with high probability. We propose the following procedure as one possible refinement that is somewhat less efficient and harder to implement. We pick a small random subset of all word-ids with size large enough so that the f most frequent words are contained with high probability (f is small here, for instance 3 or 4). The probability that the i-th most frequent word-id is in a random subset of size s is equal to

$$1 - \left(1 - \frac{\Omega}{i}\right)^s \approx 1 - \exp\left(-\frac{\Omega}{i} \cdot s\right) \tag{8.3}$$

To ensure that all of the f words are in the subset with high probability, we should pick s such that $S \gg f/\Omega$ (e.g., for $S > 5 \cdot f/\Omega$, Equation 8.3 has value that is already larger than 0.99). Say $W_1, ..., W_{f_1}$, where $f_1 \geq f$, are the distinct words from the subset. We reserve the first f_1 ranks and allocate an array $Count[1..f_1]$ (initially set to 0). We maintain a mapping $g : x \mapsto i$, for $x = W_i$ and $g : x \mapsto -1$ for $x \neq W_i$ so that we can count the frequencies of $W_1, ..., W_{f_1}$ by increasing $Count[g(W_i)]$ as we go through the list and compute the MTF transform. At the end we assign the ranks of $W_1, ..., W_{f_1}$ computed by sorting their frequencies and keep the ranks obtained by the MTF transform for the rest of the word-ids. The latter can be done by maintaining another mapping $r : x \mapsto$ rank of x for $x = W_i$ and $r : x \mapsto -1$ for $x \neq W_i$. We note that all computations required for this algorithm are done in-memory and no posting is required to be revisited on disk.

8.4.3. Algorithm Engineering of the In-memory Inversion

Our algorithm uses two-level posting accumulation and assigns dynamically growing arrays or bit-vectors to each group of HYB blocks. The doc-ids and the positions are compressed on the fly while the word-ids are accumulated without compression and compressed right before being written to disk.[2] Elias-gamma code is

[2] The word-ids must be stored before compression since they are entropy-encoded.

Table 8.6.: Rate given in MiB/s of our MTF zipf compression (without the proposed refinement at the end of the section) compared to zipf compression with sorting and gap encoding with Elias-gamma code for compressing positions (taken as a reference).

Compression	Zipf with sorting	Zipf with MTF	Gap + Elias gamma coding
Rate	135 MiB/s	343 MiB/s	155 MiB/s

used to compress the doc-gaps, position-gaps and word-frequencies, and Zipf compression from Section 8.4.2 to compress the word-ids. We note that the groups of HYB blocks are still well compressed because, as shown by Bast and Weber [2006], the HYB index with positional information has smaller empirical entropy[3] than that of the inverted index for any choice of block size. Once the memory limit of the in-memory HYB blocks has been reached, the groups of HYB blocks are processed one by one by decompressing each group and restoring and (re)compressing the individual HYB blocks. To encode the document and position gaps in the HYB groups we use variable-byte encoding due to its speed.

To maintain the word-to-word-id mapping, a fast hash-table based vocabulary [Heinz and Zobel, 2002] is employed (an alternative to permanent vocabulary is discussed in Section 8.7.5). To determine the corresponding HYB block for each posting, a data structure is employed that computes the HYB block efficiently as follows. For each 3-gram, the interval in the lexicographically sorted list of word boundaries that contain that 3-gram as prefix is precomputed and stored. Whenever a new posting comes, its prefix is looked up and if the resulting interval contains more than one word, a binary search is performed to find the true word boundaries (the intervals usually contain small number of boundary words, for example the most frequent 3-letter prefix in the 2000 boundary words of the GOV2 collection was `int` with 11 occurrences. All other 3-letter prefixes typically occurred less than 3 times). The newly computed ID of the HYB block is then stored in the word's vocabulary entry so that the computation is done only once per each distinct word.

The single-pass approach by Heinz and Zobel [2003b] must sort the index words whenever a new compressed run is being written out to disk as otherwise merging of the temporary inverted lists would not be possible. Note that while the HYB construction does not require word sorting, the word boundaries are already precomputed in fixed (lexicographic) order. Moreover, the postings that correspond to each HYB block are already in doc-id order and do not require sorting either.

8.5. On-disk HYB Inversion

As we have discussed in the related work section, index construction usually operates in two passes. In the first pass each collection is divided into n parts, where the size of each part is defined by the available main memory. Then a partial in-memory index is created, which, once the memory limit is reached, is transfered contiguously to disk. The price for contiguous disk access in the first pass is paid in the second pass where the on-disk inversion takes place. At this point the partial indexes must be merged together through n-way merge to obtain the final index. In the following we will show that sacrificing contiguous disk access in order to avoid index merging usually pays off on the HYB index.

An advantage of the HYB index is that the HYB blocks are of relatively similar sizes and reasonably large in number. This allows us to construct the HYB index in a single pass as follows. In an initial step we compute an estimate for each block size by running an in-memory version of our algorithm on an already sampled set of documents (see Section 8.3.1). Next, we create the index file and reserve enough disk space for each block. Once the available memory is used up, we process the HYB blocks one by one, seeking to the disk location of the current partial block and writing it in-place. The index construction finishes once the entire collection has been scanned. Note that we do not pay the full cost for each disk seek since the seek time depends on the track distance [Popovici *et al.*, 2003]. Our algorithm makes jumps with comparably similar distances compared to the size of the index, and hence keeps the track distance small, e.g., about 10 MB in average for a compressed index of size 20 GB. Table 8.7 shows this empirically. A subtlety here is a slight inherent non-linearity of the disk cost as the seek time slightly increases with the index (collection) size (which is reflected in our results).

[3] The empirical entropy defined in Bast and Weber [2006] estimates the inherent space complexity of the HYB and the inverted index independently of the specialties of a particular compression scheme.

Table 8.7.: Average random and equidistant seek time over 2000 disk seeks for different file sizes (the distance in the second row is 1/2000-th of the file size.

File size	100 MB	1 GB	10 GB	100 GB
Average random seek time	1.5 ms	5.2 ms	11.1 ms	14.5 ms
Average equidistant seek time	0.8 ms	4.7 ms	5.6 ms	6.2 ms

This behavior, however, does not affect the running times seriously on collections of the order of TREC GOV2.

The one potential problem of this approach are the blocks that require more than the estimated space. A straightforward solution is to simply leave enough free space after each HYB block on disk. Another simple solution is to use a separate file for each HYB block. This solution will, however, increase the construction cost as well as the block reading cost at query time since the operating system usually does not guarantee that the physical blocks are indeed laid out contiguously on disk. A third solution that does not affect the indexing performance significantly is a procedure that we call *space propagation*, described in detail in Section 8.5.3.

Note that in-place writing is impossible to achieve on the inverted index, first, because most of the inverted lists are short and their size cannot be reasonably well estimated and second, because the number of inverted lists is in the order of millions, making in-place list writing not worthwhile.

8.5.1. Disk Cost of In-place and Merge-based Inversion

In the following we show and experimentally confirm in Section 8.7, that in-place HYB index construction is more efficient than the standard paradigm of two or three passes over the data depending on whether the merge is *in-situ* or not. Let C and U be the collection size in compressed and uncompressed format respectively. Let R_x be the cost of sequentially reading x bytes, W_x be the cost of sequentially writing x bytes, T_s be the disk seek time and B be the size of the in-memory buffer used for in-memory posting accumulation as well as index merging (reducing the number of disk seeks by allocating a part of the buffer to each on-disk run). The total disk cost for merge-based index construction is then roughly equal to

$$R_u + \left(R_c + W_c + \left(\frac{C}{B}\right)^2 \cdot T_s\right) + (R_c + W_c) + W_c \tag{8.4}$$

$$= R_u + 3 \cdot W_c + 2 \cdot R_c + \left(\frac{C}{B}\right)^2 \cdot T_s \tag{8.5}$$

The second and the third terms in the left-hand side correspond to the cost to merge and permute the merged file. The last term in the right-hand side corresponds to the disk seek cost, where C/B is the total number of runs. If twice more disk space than the size of the index file is allowed (the merging is not *in-situ*), then the third term should be skipped (note that our approach requires almost no additional disk space, see Section 8.7.4). Let k be the total number of HYB blocks and B be the buffer size for in-memory posting accumulation. The total disk cost for the in-place HYB construction is then equal to

$$R_u + \left(W_c + k \cdot \frac{C}{B} \cdot T_s\right) \tag{8.6}$$

since the collection should be read and written in compressed format only once. The second term in the brackets corresponds to the total seek time since each block requires a single disk seek for each run. Even though the number of disk seeks is increased by a factor of $k \cdot B/C$, the disk transfer cost $2 \cdot (W_c + R_c)$ is usually the dominating part in Equation 8.4.

Example 8.5.1. *Consider a 50 GB collection with 20 GB of compressed (positional) index. Let the disk throughput be 50 MB/s and let a single disk seek take 5 ms. Let the memory limit in both cases be 1024 MB (20 runs). For* $k = 2000$ *the disk costs for INV and HYB are 3072 and 1633 secs, respectively. Furthermore, the disk seeks take around 14% of the total disk cost of the HYB construction.*

One way to reduce the number of disk seeks required for in-place index construction is by either allowing a larger memory buffer or by using a better in-memory compression (e.g., Golomb code instead of Elias-gamma).

In the next section we propose a procedure called *partial block flushing* that can additionally reduce the number of disk seeks.

8.5.2. Partial Block Flushing

To make room for further incoming postings, our basic in-place index construction flushes each HYB block to disk once the memory limit has been reached. Hence $C/B \cdot k$ disk seeks in total or k disk seeks per run are required. In this section we show how the total number of disk seeks required can be reduced by postponing the flushing of a certain set of blocks while flushing the rest. To achieve this, we use the observation that those blocks that correspond to stop-words are significantly larger than the rest of the blocks. For example, typically around 20% of the largest HYB blocks take half of the total memory. The idea of partial flushing has been already used in Büttcher and Clarke [2008] for speeding up index update operations. In this section we provide an analysis of the effectiveness of the procedure for a given discrepancy between the block sizes.

Our partial block flushing works as follows. All blocks are initially divided into a set of large and a set of small blocks. Let L_1 be the number of large blocks and L_2 the number of small blocks, where $L_1 + L_2 = k$. Once the memory limit has been reached, all large blocks are transfered to disk and the memory freed while all small blocks are left untouched. Whenever the size of the small blocks reach a fraction $T < 1$ of the memory, all blocks are transfered to disk and the entire memory is freed. To analyze the number of disk seeks required, assume that the small blocks occupy a fraction $f \leq T$ of the total memory and let $m = k/L_1$. Then it is not hard to show the following

Lemma 8.5.2. *Partial flushing reduces the total number of disk seeks required for constructing a HYB index by at least a factor of*

$$\frac{1}{2} \cdot \frac{m + f \cdot m}{1 + f \cdot m}$$

Proof. The relatively straightforward proof can be found in the appendix. □

Clearly the advantage of partial flushing increases when the number of large blocks is small (large m) and when the small blocks take only little memory (small f).

Example 8.5.3. *If* $f = 10\%$ *(the short blocks occupy 10% of the memory) and the number of large blocks is 10% of the number of the small blocks, the number of disk seeks will be reduced by a factor of* $4.95 \cdot L_2/1.6 \cdot L_2 \approx 3$.

The gap between the large and the small blocks in practice is less extreme. The improvement that we achieved due to partial flushing on our test collections ranges from around 25% (for $k = 2000$) to around 40% (for $k = 10000$) less disk seeks on TREC GOV2 and Wikipedia and from around 31% to around 50% less disk seeks on DBLP.

8.5.3. Avoiding Block Collisions by Space Propagation

We already mentioned the problem that blocks with initially underestimated sizes will overflow and overwrite the neighboring blocks. Note that due to estimation errors, half of the block sizes in average will be underestimated. Also note that once the block writing starts, further movement of the blocks is not possible. A solution to this problem that does not affect the query and the indexing performance is to allow the underestimated blocks to make use of the space of the overestimated blocks without splitting them in two parts. Below we give a description of the procedure and provide theoretical evidence that it fails only with small probability given that certain assumptions are satisfied.

The idea is to shift the unused space of the overestimated blocks towards the underestimated ones by permitting an underestimated block to borrow space from its neighbor. If the lending block is not large enough to fit its own data plus the additional data of the borrower, it becomes a borrower, too (of the next neighbor). Hence, a cascade of blocks that borrow space from their neighbors can be formed. The cascade terminates once a large enough block to amortize for the propagated space request is encountered (note that the propagated space along the cascade of blocks can increase or decrease, see Figure 8.4).

To determine if a certain block will overflow or not, after a significant fraction of the data has been processed, each block size is re-estimated every time a new run is being written to disk. After each re-estimation the newly

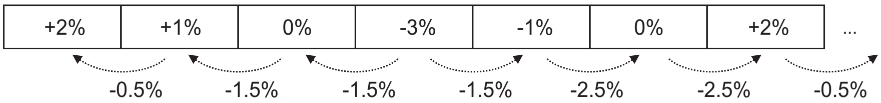


Figure 8.4.: Space propagation. The rectangles corresponds to blocks, where the percentage inside the block is the estimation error for that block (“+“ indicates overestimation and “-“ indicates underestimation). Below is the propagated space of the blocks.

estimated block sizes are compared to those that have been initially estimated. If the estimated block size is less than the space allocated for that block, then the left or the right boundary of the block is moved towards the next block and the block is extended for the size of the current run. Note that this does not require actual moving of blocks on disk, but merely writing the new run before or after the block.

The space propagation at certain block fails if both neighbors of that block have free space that is less than the required. Note that instead of writing the runs at the beginning of the block, the runs are written in the middle of the allocated space and alternated from left to right so that both sides of the block grow equally fast. This requires defragmenting the blocks once they are read from disk by decompressing each run separately. Also note that even though the blocks are of roughly the same size, there is a possibility that a certain block is too small for the space propagation demand of its larger neighbors. To address this, all blocks are initially sorted with respect to their estimated size. This increases the chance that neighboring blocks are of roughly the same size and will not interrupt the propagation.

Observation 8.5.4. *Under certain assumptions, the event of one or more space propagation failures occurs with small probability (the assumptions as well as the calculation of this probability is given in the appendix).*

There are two options in case of space propagation failure: the affected block could be split into two parts by writing the remaining part of the block to a different location; or the affected block could be entirely relocated at the end of the index file, thus providing extra free space for its neighbors. The second alternative requires touching a small number of on-disk postings for the second time. This event, however, occurs with small probability. Furthermore, we limit the number of block relocations to a small constant (e.g., 5 blocks) which amounts to less than 1% of the total number of postings.

8.6. Constructing a Fuzzy Index

In this section we describe the modifications required to achieve fuzzy search. Recall that our fuzzy index consists of fuzzy inverted lists. Unlike an ordinary inverted list, a fuzzy inverted list comprises the list of postings of a precomputed set of words or the list of postings of a set of words that are completions of a precomputed set of prefixes. A HYB block, on the other hand, comprises the list of postings of a precomputed range of words. In the following, we will refer to both, word clusters and $*$-prefixes simply as clusters. To represent the fuzzy inverted lists by HYB blocks, each cluster is represented by a unique id and mapped to an artificial word range. For example, the following word-range corresponds to a cluster with id 101:

```
C:101:aglorithm
C:101:algorithm
C:101:algoritm
...
```

The clustering is represented by a data structure initially precomputed from the dictionary data of the corpus and stored in a separate file. For each word, this file contains a list of cluster ids to which it belongs, for example

```
algorithm 0, 101
algorithmica 0, 105, 2011
algoritm 0, 101, 105, 405
...
```

Say the set of clusters with ids 10, 319 and 1012 represents a cover for some keyword q given at query time. Then q is represented as `C:101:*|C:319:*|C:1012:*`, where `|` and `*` are the disjunction and the prefix operators. The index construction goes in three steps. First, an initial pass is performed in order to compute the dictionary and the term frequencies, which in turn are required to compute the clustering (see Sections 5.3.3 and 5.4.3). Since an initial pass is performed, the exact sizes of each HYB block are known in advance. In the second step, the clustering is computed and stored on disk in a separate file. This file is used during both, index and query time. In addition, a fuzzy word or prefix matching data structure is precomputed and stored on disk along with the clustering file. This prevents constructing this data structure from scratch whenever the index is used. The actual index is constructed in the third step. In what follows, we give the main differences compared to constructing a HYB index.

8.6.1. Posting Accumulation

Recall that each cluster id corresponds to a HYB block. Since this information is already precomputed and available, it is initially read and stored for fast access in the hash-table along with the word's id. Hence, instead only to one, every posting from the parsing stream is appended to each of its HYB blocks during the posting accumulation.

8.6.2. On-Disk Inversion

Two further important differences are that, first, the total number of HYB blocks is usually larger compared to a HYB index for exact search, and second, there are HYB blocks with comparatively smaller sizes (those that correspond to clusters with rare words). We set a limit on the size of the blocks as done in Section 8.5.2 and divide them into large and small blocks. The large blocks are then written in-place as in Section 8.5. The small blocks are written contiguously in runs in a separate temporary file. At the end of the index construction this file is merged and appended to the main index file.

8.7. Experiments

We compare an in-place and a merge-based variant of our HYB construction for exact prefix search, as well as a variant that supports fuzzy prefix search, against our own implementation of the state-of-the-art inverted index construction from Heinz and Zobel [2003b], (including the cache-friendly two-level posting accumulation from Section 8.4.1 as an additional optimization). As a reference, we also compare against the authors' implementation of the same algorithm that comes as a part of the Zettair[4] search engine. The index construction of Zettair essentially implements the ideas from Heinz and Zobel [2003b] but slightly varies from the original single-pass approach by using log-10 partitioning instead of a single n-way merging. According to a large study of Middleton and Baeza-Yates [2007], Zettair's index construction is indeed the fastest on the open source market to date, with performance close to twice as fast as the second fastest option.

8.7.1. Experimental Setup

Our implementation of the HYB index construction, as described in Section 8.3 and 8.4, writes all blocks to a single file, with an array of block offsets at the end of the file. The vocabulary is compressed with zlib and stored in a separate file on disk. All our code is written in C++ and compiled with GCC 4.1.2 with the -O6 flag. We used the latest (0.9.3) stable version of Zettair.

The experiments were performed on a machine with 4 quad-core AMD Opteron 2.8 GHz processors with 8 MB cache, operating in 64-bit mode on a fast RAID file system with sequential read/write rate of up to 600 MiB/s.

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

Table 8.8.: Summary of the three test collections used in our experiments.

Collection	Raw size	Index size	Documents	Occurrences	Distinct words
DBLP	1.1 GB	0.3 GB	0.3 millions	0.15 billions	1.2 millions
Wikipedia	21 GB	7.2 GB	9.3 millions	3.2 billions	29 millions
TREC GOV2	426 GB	46 GB	25 millions	23 billions	57 millions

Table 8.9.: Elapsed index construction time given in minutes to construct a word-level index with our HYB builder and Zettair on Wikipedia and the TREC GOV2 collection.

	Wikipedia	25% GOV2	50% of GOV2	100% of GOV2
HYB (in-place)	11 min	21 min	36 min	70 min
HYB (merge)	13 min	27 min	45 min	90 min
HYB (fuzzy)	31 min	49 min	91 min	148 min
INV	19 min	34 min	57 min	146 min
Zettair	23 min	35 min	70 min	127 min

We investigate the impact of hard disk speed on index merging and in-place block writing from Section 8.5 separately by carrying out experiments on two additional hard disks: a relatively fast (Seagate Savvio) hard disk with 10000 RPM, sequential read/write rate of around 150 MB/s and average reported access time of 4.6 ms (HDD1); and a slow Samsung hard disk, with 7200 RPM, sequential read/write rate of 50 MB/s and average access time of 20 ms (HDD2).

For both algorithms, we imposed the same memory limit for the in-memory inversion. We used the `-big-and-fast` option on Zettair which allows 500 MB of memory. According to Heinz and Zobel [2003b] a higher memory limit does not significantly impact the performance of their construction algorithm. We did not take into account the additional memory usage of the vocabulary and other auxiliary data structures. To construct a HYB index for exact prefix search, we picked the HYB block sizes as $N/5$, where N here is the number of documents in the collection (see Bast and Weber [2006]). To construct a HYB index for fuzzy prefix search, we picked $*$-prefixes of length 4, with 1 “don’t care“ character that resulted in space overhead between factor of 2 and 3 (Section 5.4).

To ensure that both parsers produce an equal number of word occurrences, we replaced each non-alphanumeric character in the collections by a single space.

8.7.2. Test Collections

Our experiments were carried out on three collections:

DBLP: a selection of 31,211 computer science articles with a raw size of 1.3 GB, 157 million word occurrences (5,030 occurrences per document in average) and 1.3 million distinct words. Our goal was to investigate the impact of the construction algorithm on collections that are small and can be fully inverted in main memory.

WIKIPEDIA: a dump of the English Wikipedia, with a raw size of 21 GB, 9,326,911 documents, 3.2 billion word occurrences (343 occurrences per document in average), and a vocabulary of 29 million distinct words.

TREC GOV2: the TREC GOV2 (or TREC Terabyte) corpus with a raw size of 426 GB, 25,204,103 documents, 23 billion word occurrences (912 occurrences per document in average), and a vocabulary of 57 million distinct words. To get a better picture on the scalability of the algorithms we ran the experiments on subsets of size 25%, 50% and 100% of the full collection.

8.7.3. Index Construction Time

Table 8.9 shows the index construction running times on two of our test collections. Compared to our own INV construction, the construction of a HYB index for exact search is faster by a factor of 1.7 (on Wikipedia) to a

Table 8.10.: Break-down of the total elapsed index construction time by 5 subroutines on the TREC GOV2 collection.

Parsing & Hashing	Accumulation	Compression	Disk I/O	Sampling
32%	16%	20%	29%	3%

factor of 2 (on GOV2). Compared to Zettair, we are faster by a factor of 1.7 - 2 (on GOV2) to more than a factor of 2 (on Wikipedia). The merge-based version of our HYB construction algorithm is slower than the in-place version by a factor of 1.2 (on Wikipedia) to a factor of 1.3 (on GOV2). The difference on the DBLP collection was small and it is not shown.

Constructing a HYB index for fuzzy search takes from 2 (on GOV2) to 3 times longer (on Wikipedia) compared to constructing a HYB index for exact search, which is about equal compared to constructing an inverted index on a sufficiently large corpus. The larger running times are not surprising since an additional pass is required as well as processing twice to three times as many postings (accumulating, compressing and writting to disk).

We point out that the advantage on a particular system depends on the ratio of disk and CPU speed. The advantage on systems with high disk bandwidth is less dramatic since our improvement is slightly larger with respect to disk time than with respect to CPU time.

Posting Accumulation

Table 8.10 shows a breakdown of the HYB construction time by 5 major subroutines. Only about 16% of the time it takes to construct our index is spent on posting accumulation or in-memory inversion. As suggested in Section 8.4.1, for the inverted index construction this cost is more than twice as much. This suggests that the cache inefficiency seriously affects the running time of the in-memory inversion of the single-pass approach. For whatever reason, posting accumulation cost is not reported in the elapsed-time figures of Heinz and Zobel [2003b].

In-place vs Merge-based Index Construction

Index merging for the HYB index is simple to implement and requires no priority queue to find the next chunk of data to be written to disk. It works by allocating a buffer to each on-disk run and writing the partial HYB blocks in-order by iterating the buffers one by one. Whenever empty, each buffer is refilled with the next few partial blocks from the corresponding run.

Figure 8.5 compares the disk costs of merge-based versus in-place index construction on two different hard disks. In-place block writing almost always outperforms index merging with the difference between the two being larger on the faster hard disk where a disk seek is less expensive. Index merging is faster on the slower disk only when the available main memory is very low.

While merge-based (INV and HYB) index construction spends about 20% - 30% of the total time on writing and merging the index, only about 5% of the time on the faster hard disk and about 15% of the time on the slower hard disk is spent on in-place block writing. The reason for this is that merge-based construction has to fully read and write the temporary index file more than once. We note that the actual bottleneck of index merging is not caused by the disk seeks (in our experiments the disk seeks took about 1% of the HYB merging time and below 15% of the INV merging time) but is rather the result of reading and writing large amounts of data from disk. As a consequence, the index partitioning improvement proposed in Heinz and Zobel [2003b] which aims to reduce the number of disk seeks by performing an in-memory index merge, has been only marginally faster in practice.

In Section 8.5 we saw that the disk seek cost of our construction algorithm depends on the number of blocks as well as on the total number of runs which in turn depends on the main memory limit. Table 8.11 shows the increase in running time of the HYB index construction for varying number of HYB blocks when two different memory limits are used. An increase by a factor of 10 in the number of blocks increased the total running time by about 8% when the memory limit was set to 1024 MB and by about 21% when the memory limit was set to 512 MB.

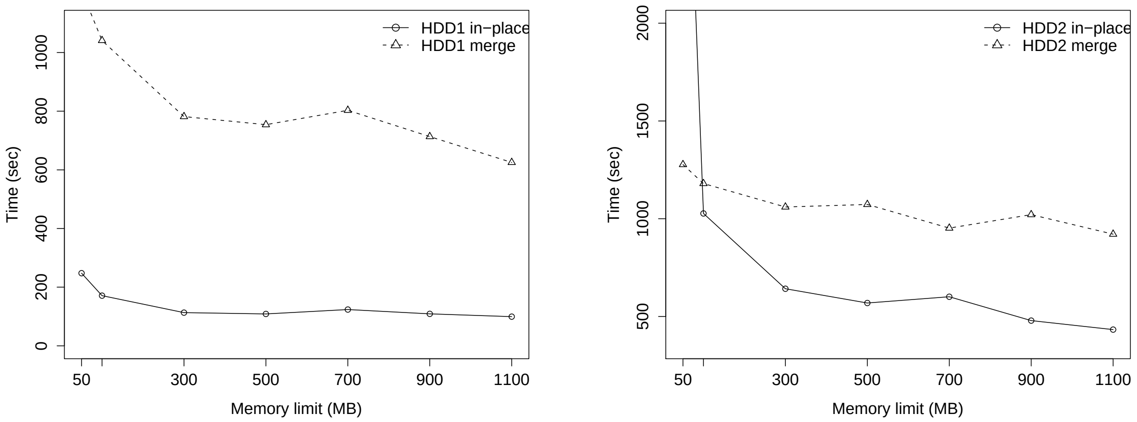


Figure 8.5.: Disk cost of in-place and merge-based HYB index construction on the GOV2 collection, when a fast (left) and a slow hard disk is used (right) (see Section 8.7.1).

Table 8.11.: Increase in HYB construction time given in percents for varying number of HYB blocks and two different memory limits. The experiment was carried out on the GOV2 collection.

Mem. Limit	2000 blocks	4000 blocks	6000 blocks	8000 blocks	10000 blocks
512 MB	0%	+2.9%	+9.2%	+13.3%	+21.5%
1024 MB	0%	+1.6%	+5.8%	+6.5%	+8.2%

Parsing and Hashing

The most expensive phase is the parsing and hashing of words which takes roughly one third of the total cost. We believe that there is not much room for improvement here since a *hash table with move-to-front chains* and a *bit-wise hash function* has been the fastest practical data structure for in-memory vocabulary accumulation for some time, conditioned on the assumption that the words do not need to be maintained in sort order [Zobel *et al.*, 2001]. An alternative fast and practical data structure that maintains the sorted order of the words is the *burst trie* [Heinz and Zobel, 2002].

8.7.4. Temporary Disk Space

The additional temporary disk space usage during the HYB index construction (other than that required for the final index) is very small and comes from the overflowed blocks. The total size of the overflowed space depends on the quality of the initial estimation of the block sizes (see Section 8.5.3). The additional peak disk space usage in our experiments varied from 1% of the size of the final index file on small to medium-size collections, up to 3% on the full TREC GOV2 (up to 3% overhead).

Our inverted index construction writes the index in a separate file by merging the temporary index file. The total additional space usage is hence 100% or twice the index size. The reported additional temporary disk space usage in Heinz and Zobel [2003b] is about 26% for document-level inverted indexes and about 8% for word-level inverted indexes. However, we found out that during the parsing and merging phase the peak additional space usage of Zettair on the TREC GOV2 was about 65% and 43% respectively of the size of the final index file. We note that in a setting where the input data is very large and streamed, using significantly more space than the final index might be undesirable. Table 8.12 gives a summary of the peak disk space usage (note that the total and not the additional disk usage is shown).

8.7.5. Refinements

Our basic algorithm assumes that the entire vocabulary fits into main memory. Given that the vocabulary of TREC GOV2 takes roughly 600 MB, this is a realistic assumption. Still, for the case where one wants to get rid of the assumption, we propose the following refinements.

Table 8.12.: Peak temporary disk space usage compared to the size of the final index.

	Zettair	HYB construction	INV construction
Wikipedia	163% (12 GB)	101% (8 GB)	200% (16 GB)
TREC GOV2	165% (82 GB)	103% (46 GB)	200% (108 GB)

Table 8.13.: Increase in HYB construction time on one quarter of the TREC GOV2 collection when different in-memory vocabulary size limits are used (see Section 8.7.5). The total number of distinct words was roughly 28.5 millions and the construction (without vocabulary limits) took 43 minutes.

Vocabulary size limit in %	50%	33%	25%	20%
Large Hash-keys	+7%	+10%	+12%	+11%
On-disk vocabulary	+14%	+20%	+22%	+23%

Large Hash Keys

Instead of permanently storing word - word-id pairs, we can compute the word-ids with an additional hash function, making it possible to flush the vocabulary to disk once its size approaches the memory limit. At the end of the inversion the sorted vocabulary fragments[5] are merged into a final vocabulary (without fully reading them in memory). To avoid word-id collisions one can easily show that by providing a universal family of hash-functions with large enough hash keys, the expected number of collisions can be kept below 1.

Let v be the vocabulary size and r be the size of the range of the hash function. Let X_i be the indicator random variable that is 1 iff the i-th word collides with an earlier word, and 0 otherwise. Since $Pr(X_i = 1) = (i-1)/r$, the expected number of such collisions is equal to

$$\sum_{i=1}^{v} Pr(X_i = 1) = \sum_{i=1}^{v} \frac{i-1}{r} = \frac{v(v-1)}{2 \cdot r}$$

Hence, $v(v-1)/2r < 1$ as long as $v < (1 + \sqrt{8 \cdot r})/2$ which is roughly equal to 6.074 billions for $r = 2^{64}$ (the TREC GOV2 collection has roughly 50 million distinct terms).

The running time overhead imposed by this procedure amounts to only about 10% of the total index construction time in practice (see Table 8.13). The size of the compressed word-ids does not significantly increase since the word-id compression algorithm works by compressing the word ranks instead of directly compressing the word-ids. The unique word-ids are only stored in the codebook (rank to word-id mapping), which due to the Zipfian distribution of the word-ids has small size (see Section 8.4.2).

On-Disk Vocabulary

If one wants to avoid 64-bit word-ids, an alternative approach to limit the size of the vocabulary is to keep a part of the vocabulary on disk in the following way. Once the limit of the vocabulary size is reached, a small number of (word, word-id, hash-value) triples that correspond to rare words is flushed and appended to a temporary file on disk so that the vocabulary size is slightly below the limit. Whenever a new run is to be compressed and written out to disk, each word is read from the file and checked against the hash-table. If the word is in the hash-table this means that an already flushed word has occurred again, inevitably having a different word-id. To prevent multiple word-ids for a single word, the new word-id (that is already in the vocabulary) is replaced with the initial. To address the problem of incorrect word-id assignments in the current in-memory run, instead of word-ids we store pointers to the word-ids that are stored in the vocabulary, without using compression.

The overhead in the index construction time here is about twice as large as that in the previous approach and it comes from frequently reading from disk a gradually growing fraction of the vocabulary.

[5] The vocabulary is sorted before flushing it to disk.

Multiple Cores

Another way to address the vocabulary size problem is to extend our algorithm to work in a multi-processor environment as follows. Let C be the number of processor cores. The first easy-to-implement approach simply assigns a group of k/C HYB blocks to each core, where one of the cores also does the parsing. Hence the hashing, compression and posting accumulation part of the inversion (which in total accounts for more than 1/2 of the total time) will benefit from a speed-up by a factor of up to C. Because each of the cores is responsible for only a certain range of words their vocabulary sizes remain reasonably small to fit in main memory.

A second approach that is orthogonal to the first and exhibits more parallelism, makes use of the observation that most of the words in large vocabularies are rare. If the TREC GOV2 collection is split into 16 equal-sized segments, the total number of words in each of the 16 vocabularies will sum up to only twice the size of the vocabulary of the original collection. Instead of a group of HYB blocks, each core is now responsible for a fraction of all documents, without a central core. To prevent multiple word-ids, the cores should use a common hash-function to generate the word-ids (as in Section 8.7.5).

If a single index is desired then each core must write its run in the same index file. Since this can happen at any given point in time, either the run writing should be synchronized (the first core is always the first to write, the second core always the second, etc.) or additional header data for the order of runs should be included in the blocks. The time needed for the whole in-memory part of the inversion, which in total accounts for more than 2/3 of the total index time, will be reduced by a factor of up to C.

In case of multiple indexes, each core has its own index and one of the cores in addition is in charge for answer-set merging across the partial query results.

9. Snippet Generation

In this chapter, we describe an algorithm for efficient generation of textual snippets when the query words non-literally match the documents, as those arising from fuzzy search. Snippet generation is typically implemented by searching, at query time, for occurrences of the query words in the top-ranked documents. We will refer to this approach as the *document-based* approach. Non-literal matches are an inherent problem of the document-based approach since localization of the corresponding snippets becomes problematic. This is especially pronounced for fuzzy prefix search where the number of matching words per keyword can be very large. The problem is aggravated when advanced query operators are used (e.g., phrase or proximity search) since each query operator has to be implemented twice, once on the index side in order to compute the result set, and once on the snippet generation side to generate appropriate snippets. Moreover, in the worst case, the document-based approach requires scanning of the whole document for occurrences of the query words, which is problematic for very long documents.

We propose a novel *index-based* method that localizes snippets by information solely computed from the index that overcomes all three problems. Unlike previous index-based methods, we show how to achieve this at essentially no extra cost in query processing time, by a technique we call *operator inversion*. We also show how our index-based method allows the caching of individual segments instead of complete documents. This enables a significantly larger cache hit-ratio compared to the document-based approach, which, in turn, is the key to high-performance snippet generation.

In Section 9.1, we introduce the document and index-based snippet generation in finer detail and give a more detailed overview of the contribution in this chapter. In Section 9.2, we provide a short survey of the related work. In the following Sections 9.3, 9.4 and 9.5, we provide a detailed step-by-step implementation of our index-based approach. In Section 9.6, we present our segment caching and compare its effectiveness to the standard document-based cache, both experimentally and theoretically. In Section 9.7, we provide experimental evaluation of our approach and state-of-the-art document and index-based approaches.

9.1. Introduction

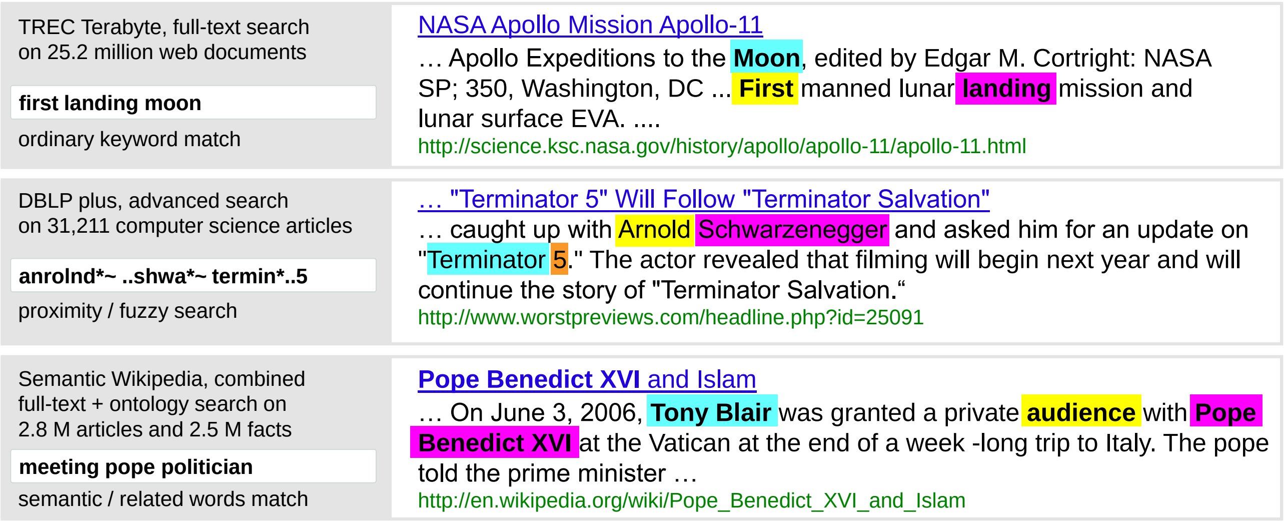
Ranked result lists with query-dependent snippets, as shown in Figure 9.1, have become state of the art in text search. Previous work on snippet generation has mostly focused on the quality aspect: which snippets should be displayed and why. The focus of this work is on the efficiency aspect: how to compute good snippets as fast as possible under non-literal matches that arise in search application such as fuzzy search. In addition, we consider an important engineering aspect: how to avoid the code duplication that is usually associated with implementation of snippet generation for more complex query operators. Hence, the contribution of this chapter extends well beyond fuzzy search.

9.1.1. Document-based Snippet Generation

In Section 9.2 we survey the existing methods for query-dependent snippet generation that have been described in the literature and implemented in (open source) search engines.[1] These methods differ in which of the potentially many snippets are extracted and shown, and in how exactly documents are represented so that snippets can be extracted quickly. On a high level, however, the bulk of them follow the same principle two-step approach from Turpin *et al.* [2007], which we call *document-based*. In the following description and throughout the paper we will assume a precomputed partitioning of the documents into *segments* of bounded lengths. Although it is irrelevant to our approach how this partitioning is done, we will assume that each segment corresponds to a sentence.

The document-based snippet generation can be described by the following principle steps.

[1] We can only guess how snippet generation is implemented in commercial products or in the big web search engines.



TREC Terabyte, full-text search on 25.2 million web documents

first landing moon

ordinary keyword match

NASA Apollo Mission Apollo-11

… Apollo Expeditions to the **Moon**, edited by Edgar M. Cortright: NASA SP; 350, Washington, DC ... **First** manned lunar **landing** mission and lunar surface EVA. ....

http://science.ksc.nasa.gov/history/apollo/apollo-11/apollo-11.html

DBLP plus, advanced search on 31,211 computer science articles

anrolnd*~ ..shwa*~ termin*..5

proximity / fuzzy search

… "Terminator 5" Will Follow "Terminator Salvation"

… caught up with Arnold Schwarzenegger and asked him for an update on "Terminator 5." The actor revealed that filming will begin next year and will continue the story of "Terminator Salvation.“

http://www.worstpreviews.com/headline.php?id=25091

Semantic Wikipedia, combined full-text + ontology search on 2.8 M articles and 2.5 M facts

meeting pope politician

semantic / related words match

Pope Benedict XVI and Islam

… On June 3, 2006, **Tony Blair** was granted a private **audience** with **Pope Benedict XVI** at the Vatican at the end of a week -long trip to Italy. The pope told the prime minister …

http://en.wikipedia.org/wiki/Pope_Benedict_XVI_and_Islam

Figure 9.1.: Three examples of query-dependent snippets. The first example shows a snippet (with two parts) for an ordinary keyword query; this can easily be produced by the document-based approach. The second example shows a snippet for a combined fuzzy prefix search / proximity query (~ is the fuzzy search operator, * is the prefix operator and `..` is the proximity operator). In particular, the document-based approach needs to take special care here to display only those segments from the matching documents where (non-literally matching) spelling variants of the keywords indeed occur close to each other. The third example shows a snippet for a semantic query with several non-literal matches of the query words, e.g., Tony Blair matching politician. As explained in Section 9.1.3, processing of this query involves a join of information from several documents, in which case the document-based approach is not able to identify matching segments based on the document text and the query alone. The index-based approach described in the paper can deal with all three cases without additional effort. For ordinary queries it is at least as efficient as the document-based approach.

(D0) For a given query, compute the ids of the top ranking documents using a suitable precomputed index data structure.

(D1) For each of the top-ranked documents, fetch the (possibly compressed) document text, and extract a selection of segments best matching the given query.

Note that we call the first step D0 (instead of D1) because it is not really part of the snippet generation but rather a prerequisite. The same remark goes for I0 below.

9.1.2. Index-based Snippet Generation

The index-based approach to snippet generation can be described by the following principle steps.

(I0) For a given query, compute the ids of the top-ranking documents. This is just like (D0).

(I1) In addition to the information from (I0), also compute, for each query word, the matching positions (postings) of that word in each of the top-ranking documents.

(I2) For each of the top-ranked documents, given the list of matching postings computed in (I1) and given a precomputed segmentation of the document, compute a list of positions to be output.

(I3) Given the list of positions computed in (I2), produce the actual text to be displayed to the user.

The reader might wonder why we need an additional step (I1) here, given that this information is already contained in a positional index. However, in Section 9.3.1 we show that efficient incorporation of this information in a result list is far from trivial.

9.1.3. Document-Based versus Index-based

In this section, we argue indexed-based snippet generation is superior to document-based snippet generation in several practically relevant respects.[2]

Non-literal matches

One of the central problems in text retrieval is the vocabulary mismatch problem, that is, when a relevant document does not literally contain (one of) the query words. Although our original motivation is fuzzy search, there is a variety of other advanced search features which call for non-literal matches, including: related-word or synonym search (as just explained), prefix search (for a query containing `alg`, find `algorithm`), semantic search (for a query `musician`, find the entity reference `John Lennon`), etc. A nice list of commented examples from the TREC benchmarks is given in Buckley [2004].

A common way to overcome the vocabulary mismatch problem in general is to index documents under terms that are related to words in the document, but do not literally occur themselves. For example, the fuzzy inverted list of the word `probabilistic` typically contains hundreds of spelling variants also in documents that do not contain `probabilistic` literally. When such a document is returned as a hit for a query containing `probabilistic`, it is, of course, desirable that a segment containing the misspelling `probablistic` is displayed as one of the snippets. The document-based approach needs additional effort to achieve this. An obvious extension would be to add the additional terms not only to the index but also to the documents. This is indeed how we used to realize this feature in our own search engine prototype in the past. This extension has several disadvantages: First, it complicates and slows down the procedure for extracting matching segments from a document. Second, it blows up the space required for each document, and, for large documents, can also affect the snippet generation time in case the document is fully read. Third, inserting additional terms into documents after the parsing, as required in many cases, is hard. Fourth, this is an instance of code duplication, as discussed as a separate issue below.

The index-based approach does not have any of these problems. After the top-ranking documents and corresponding positions have been computed in steps (I0) and (I1), steps (I1) and (I2) are completely oblivious of what the query words are, and they only deal with positions.

Code duplication

A major system's engineering problem of the document-based approach is that each and every search feature which defines what constitutes a hit has to be implemented twice: once for computing the ids of the best matching documents from the index, and then again for generating appropriate snippets.

For example, consider the fuzzy search query from Figure 9.1. The fuzzy search and proximity operator need to be considered in step (D0), so that only those documents are considered as hits where spelling variants of `anrolnd` and fuzzy completion of `shwa` occur in proximity of each other (e.g. `arnold schwarzenegger`) together in documents where exact completions of the prefix `termin` and the number `5` occur in proximity of each other (e.g. `tirminator 5`). Then the same operators need to be considered again in step (D1), so that we only extract (or at least prefer) snippets where the appropriate words occur in proximity of each other.

Since (D1) does not have access to the positional information used in (D0) — that is exactly the difference between the document-based and the index-based approach — very similar functionality has to be implemented twice. This is time-consuming, error-prone, and harder to maintain.

The same holds true for *any* query operator. This includes simple ones like: phrase (require that two query words are adjacent), proximity (like in our example query), disjunction, as well as more complex ones like the join operator described in Bast *et al.* [2007] (find all authors which have published in both SIGIR and SIGMOD).

The index-based approach does not have any of these problems. Every query operator has to be implemented once, on the query processing side, such that document ids and positions are computed which satisfy the requirements of the respective operator. After that, (I2) and (I3) will compute only with those valid positions.

[2] Our argument here assumes that some kind of positional index is available; see Section 9.3. In scenarios where only a non-positional index is available, the document-based approach is without alternative.

Large documents

A less severe but still relevant problem of the document-based approach are very large documents. Since step (D1) has no access to the positions of the query words, in the worst case the whole document has to be scanned to find all required occurrences of the query words. A possible remedy would be to build a small index data structure for every document, in order to be able to find occurrences of query words more efficiently. In fact, the Lucene search engine does exactly this; see Section 9.2. But that incurs additional space and additional implementation work. In fact, it would again mean re-implementing functionality which, in principle, has already been realized for the index computation in step (D0).

The index-based approach does not have that problem, provided that we represent our documents in a way that allows efficient retrieval of those parts covering a given list of positions. As we will see in Section 9.5, a simple block-wise compression and storage scheme will do the job.

9.1.4. Our Contribution

The index-based approach is not new, however, a rigorous efficiency study was lacking so far; see Section 9.2 for a discussion of previous work. In this chapter, we provide a detailed implementation of each of the steps (I1) - (I3) that performs index-based snippet generation in negligible time, compared to the ordinary query processing in (I0). While the realizations of steps (I2) and (I3) are relatively straightforward, a straightforward implementation of step (I1) almost triples the query processing time. It also requires a modification of the central list data structure; see Section 9.3.1. Based on an idea which we call *operator inversion*, we show how to realize (I1) in time negligible compared to (I0).

We compare our implementation of the index-based approach with a state-of-the-art implementation of the document-based approach. The bottom line is that for exact matches (which are the easiest to compute for the document-based approach), without the use of caching, we achieve slightly better performance characteristics, however, with all the additional power that comes with the index-based approach; see the comparison in Section 9.1.3. We also compare our implementation against the Wumpus' [Buettcher, 2007] implementation of the well known GCL query language [Clarke *et al.*, 1995b]. Wumpus supports a variety of structural queries computed solely based on the information contained in the positional index. Our snippet generation times are an order of magnitude faster than those achieved by Wumpus. For the exact figures, see Section 9.7, in particular, Tables 9.2, 9.3, and 9.4.

As our third main contribution, we investigate the role of caching in the index-based approach compared to the document-based approach. As shown in Turpin *et al.* [2007] and re-validated in Section 9.7 of this chapter, the bulk of the snippet generation time is spent on disk seeks. Serving as many segments as possible from memory instead of from disk is therefore key to high performance snippet generation. The document-based approach dictates a very coarse caching granularity: for each document either its whole text is cached or nothing at all. This is because finding the matching segments first requires a scan over the whole document (unless approximate solutions are tolerated; see Section 9.2 and 9.7). The index-based approach naturally permits a much more fine-grained caching: step (I2) produces a request for the text of a particular segment without the need to retrieve parts of the document first. It can therefore be decided individually for each segment whether to cache it or not. We show how this leads to a significantly higher cache hit-ratio since only the most relevant snippets are kept in the cache while the rest of the documents are not considered. More specifically, this permits caching many more relevant snippets compared to document-based caching, which in turn leads to a better use of the available memory; details are given in Section 9.6 and figures in Section 9.7.4.

Finally, we have fully integrated our new method with the CompleteSearch engine [Bast and Weber, 2007], where it provides fast snippet generation for fuzzy search, but it is also compatible with CompleteSearch's other advanced query operators, in particular those which produce non-literal matches.

9.2. Related Work

Most of the work dealing with document summarization is concerned either with *query-independent summarization* Varadarajan and Hristidis [2006] or with snippet selection and ranking. Tombros and Sanderson [1998] presented the first in-depth study showing that properly selected query-dependent snippets are superior to query-independent summaries with respect to speed, precision, and recall with which users can judge the relevance of a hit without actually having to follow the link to the full document. As usually more segments match the query than could (and should) be displayed to the user, a selection has to be made. Questions of a proper such ranking have been studied by various authors. For example, Varadarajan and Hristidis [2006] have proposed an algorithm for computing segments that are semantically related to each other as much as possible. In Ko *et al.* [2007], a pseudo relevance feedback is applied to salient sentence extraction to identify the most relevant sentences. A slightly different concept in document summarization is the multi-document summarization paradigm that reduces the document size of the top-matching documents to only a few sentences by retaining the main characteristics of the original documents Wang *et al.* [2008].

Note that nowadays all the big search engines have query-dependent snippets, in particular: Google, Yahoo Search and Bing. In this paper, we take the usefulness of query-dependent result snippets for granted and focus on efficiency and feasibility aspects. We come back to the usefulness aspect in our conclusions. For the scoring and ranking of segments, we adopt the simple, yet effective scheme from Turpin *et al.* [2007], which we briefly recapitulate in Section 9.7.

Turpin *et al.* [2007] have presented a first in-depth study of the document-based approach with respect to efficiency (in time and space). They propose the following document representation: all words as well as all separating sequences are replaced by an id, where the more frequent tokens get lower ids, i.e., the most frequent term gets id 1, second most frequent id 2, etc. (very infrequent tokens remain in plain text). Ids are then universally encoded (with small ids getting a short code). In addition, they introduce document caching strategies and point out the significance of caching as a highly effective way to increase query throughput.

In a sequel of this work, Tsegay *et al.* [2009] propose a lossy document compaction strategy (so called *document surrogates*) in order to improve the effectiveness of document caching. Their approach is based on reordering sentences in a document so that sentences that are more likely to be included in a snippet appear near the beginning of the document. The remaining sentences are pruned to reduce their size and hence make a better use of the available memory. In a recent related work, Ceccarelli *et al.* [2011] use the knowledge stored in query logs to build dynamic concise document surrogates by identifying the subset of snippets in a document that is most likely to be accessed in the future (so called *suppersnippets*). They show experimentally that the number of distinct relevant snippets for a given document is in most of the cases very small. As we will see in Section 9.6 and 9.7, our caching approach does this naturally and in a sense optimally.

A variant of the index-based approach has been previously investigated in Clarke *et al.* [1995a] and Clarke and Cormack [2000]. The emphasis of this line of work is on a clean and universal query language model (based on the so-called GCL query algebra) for the retrieval of arbitrary passages of text. The authors do consider questions of efficiency, but even their most efficient algorithm touches every (positional) posting of each of the query words, which is exactly what our operator inversion, explained in Section 9.3.2, avoids. The GCL query language is implemented in the Wumpus search engine Buettcher [2007]. In Section 9.7, we will see that snippet generation times for this approach indeed outweigh the query processing times. The issue of fine-granularity caching is not discussed in this line of work.

The open-source search engines that we know of and which provide query-dependent snippet generation follow the document-based approach. In particular, this holds true for Lucene Cutting [2004], which we have studied in depth for the purpose of this work. Early versions of Lucene used a straightforward implementation of the document-based approach: at query time each of the top-ranked documents was re-parsed and segments containing the query words were extracted. To address the obvious inefficiency of this approach, recent versions of Lucene have provided support for storing the sequence of term ids output by the parser (much in the vein of Turpin *et al.* [2007]).

Another alternative is to precompute and store for each document a small inverted index for fast location of the query words. Without significant space overhead, the small inverted index will provide us the positions for each of the query words in a document. However, this does not help with avoiding code duplication for non-trivial query operators (like proximity), since it still remains to filter out the positions which obey that query operator. Simply running the query again on this index will not solve the problem, since standard query processing only provides a list of matching documents (in Section 9.3.1 we show that computing the matching postings in a straight-forward way is far from efficient). Therefore, all of the proposed enhancements remain in the realm of the document-based approach, and as such do not address the problems of non-literal matches, code duplication, and coarse caching granularity pointed out in Sections 9.1.3 and 9.1.4.

9.3. Step I1: Computing all Matching Positions

In this and the following two sections, we will describe our realization of steps (I1), (I2), and (I3) of the index-based approach as described in high-level in Section 9.1.2. We begin by presenting two algorithms for (I1), that is, computing for each query word the list of its positions in each of the top-ranked documents.

Throughout this work, we assume that we have a *positional index*, where a posting list has the structure of a fuzzy inverted list introduced in Chapter 5:

doc ids	D401	D1701	D1701	D1701	D1807
word ids	W173	W113	W94	W202	W516
positions	5	3	17	51	10
scores	0.3	0.7	0.4	0.3	0.2

A standard inverted index will have one such list precomputed for each word (that is, all word ids are the same in each such list). Given a query, the basic operation will be to either intersect the lists for each of the query words or to compute their union. We will later consider more query operators; see Section 9.3.2.

The following considerations are valid for all of these list representations and operations.

9.3.1. Extended Posting Lists

Step (I1) asks for the computation of the positions (postings) of each of the query words in each of the top-ranked documents. A positional index, as described above, obviously contains this information. A straightforward way to incorporate this information in a result list would be to augment each posting by an additional entry, telling from which query word it stems. Conceptually:

doc ids	D1701	D1701	D1701	D1701	D1701
word ids	W173	W173	W173	W179	W179
positions	5	9	12	54	92
scores	0.3	0.7	0.4	0.3	0.2
query word	1	1	1	2	2

For example, the third posting from the right, now “knows“ that it stems from the first query word (when reading a list from disk, the query word entry would be set to some special value). It is not hard to see that with this augmentation, the information required for (I1) can be computed for all of the operations described above.

There are two major disadvantages with this approach, however. The first is that we modified the central data structure, the sanctuary of every search engine. Unless the search engine was written with such modifications in mind, this is usually not tolerable. For our own research prototype, we would expect numerous undesirable side effects and bugs following such a modification.

The second major problem is efficiency. Consider an intersection of the posting lists of two query words. The extended result lists would now contain postings from both input lists, that is, it at least doubles in size compared to the corresponding simple result list. This effect aggravates as the number of query words grows. Table 9.1 below shows that this indeed affects the processing time. The problem is that per-query-word positions are computed for *all* matching documents, before the ranking is done. Note that this also happens when pruning techniques are involved, because, again, large numbers of postings (namely, candidates for one of the top-ranked doc ids) are involved in the intermediate calculations. In fact, the results from Table 9.1 pertain to such a pruning technique for disjunctive queries, following the ideas of Bast *et al.* [2006].

9.3.2. Operator Inversion

We propose to implement (I1) by what we call *operator inversion*. Besides fuzzy search, our approach works for (but is not limited to) the following query operators and their combinations: *boolean search* (AND, OR, XOR, NOT), *proximity search* (type `latent..indexing` and find documents containing the two words within a certain proximity to each other), *phrase search* (type `matrix.decomposition` and find documents containing the two words right next to each other), *prefix search* (type `decomp*` and find documents containing words

Table 9.1.: Elapsed query processing times with ordinary lists and with extended lists, and the additional time taken by the operator inversion (only step I1), on TREC GOV2 (the numbers in the third tow are in addition to the first row). The first column gives the average over 10,000 queries. The other columns provide the averages for fixed numbers of query words (description on the hardware used is given in Section 9.7).

Scheme	all	one-word	two-word	≥ three-word
Ordinary Lists	144.0 ms	17.9 ms	46.7 ms	174.9 ms
Extended Lists	342.6 ms	29.7 ms	114.5 ms	415.7 ms
Operator Inversion	+ 0.4 ms	+ 0.1 ms	+ 0.2 ms	+ 0.5 ms

starting with that prefix), *range search* (type `aida-carmen` and find documents containing words in that range), and *semantic search* (type `musician` and find documents mentioning a musician, like `John Lennon`).

We first explain our approach by an easy example that does not involve fuzzy search. We will be using the simplifying assumption that our query can be evaluated from left to right.[3] Then in Lemma 9.3.2 we will formalize our approach for arbitrary expression trees.

Example 9.3.1. *Consider the query* `efficient snippet..generation`*, which searches for documents that contain the word* `efficient` *and the two words* `snippet` *and* `generation` *in (some pre-defined) proximity to each other. Assume that in step (I0) we have already computed the set D of ids of the top-ranked documents. Let L_1 be the list of postings of the matching occurrences of* `efficient` *in D, and let L_2 and L_3 be the corresponding lists for* `snippet` *and* `generation`*, respectively. Our operator inversion algorithm then goes in two phases as follows.*

For the first phase, let I_1, I_2, I_3 be the index lists for the three query words, that is, the list of all postings of all occurrences of `efficient`, `snippet`, *and* `generation`*, respectively, in the whole document collection. Let L'_1 be the list of all postings from I_1 with a document id in D. This can be computed via a simple intersection of the two (sorted) lists of document ids. Similarly, compute L'_2 as the list of all postings from I_2 with a document id in D. For L'_3 we do something slightly different. Namely, let L'_3 be the list of all postings from I_3 with a document id d and a position i such that there is a posting from L'_2 with the same document id d and a position j, such that i and j are within the pre-defined proximity to each other. If the posting lists are sorted by document id, and postings for the same document id are sorted by position, this can also be computed with a straightforward variant of the standard intersection algorithm between I'_2 and I_3.*

This ends the first phase. Note that now L'_1 is already the desired list L_1. But all we can say about L'_2 at this point is that it contains *all the desired postings from L_2. It may also contain other postings though, namely those pertaining to occurrence of* `snippets` *in documents from D that are* not *in proximity to occurrences of* `generation`*. The last list computed in this phase, L'_3, is exactly the desired L_3 again.*

In the second phase, we compute L_i for those L'_i which are not yet L_i. In our example, this is only L'_2. We compute L_2 as the list of all postings from L'_2 with a document id d and a position i such that there is a posting from L'_3 with the same document id d and a position j, such that i and j are within the pre-defined proximity to each other. This is the same intersection operation as between L'_2 and I_3 in phase 1, except that L'_2 is now the other of the two lists. Technically, we apply the inverse *of the proximity operator now, i.e., we compute $(L'_2..L_3)^{-1}$ (which happens to be the same operator, if we define proximity as $|i - j| \leq \delta$, for some fixed δ).*

Since D is very short, each of L'_1, L'_2, L'_3, and L_2 will be very short too, and all of the involved intersections can be computed extremely fast. Indeed, our experiments in Section 9.7.3 will show that the time required for (I1) using operator inversion is negligible compared to both the time for (I0) (the query processing that has to be done anyway) and the time needed for (I2 - I3) (getting the snippet text from the documents). This ends our example.

Generalizing from the example above, let D be the sorted list of ids of the top-k matching documents for an arbitrary query with l keywords and a given k. Let $L'(q_i)$ be the posting list that contains all occurrences of the ith query word in D, for $i = 1, \ldots, l$. In the simplest case, this could be the inverted index list of q_i.[4] If the fuzzy search operator is used on q_i, then $L'(q_i)$ is a fuzzy inverted list of q_i, etc. Assume that each query operator

[3] Note that this is often the case in practice.

[4] With top-k techniques like those from Anh and Moffat [2006] or Bast *et al.* [2006] it can be a subset of this inverted list.

has an inverse operator (for example, the proximity operators and all binary boolean operators except OR are self-inverse).

As in the example above, whenever an operator is applied on two posting lists, the resulting list will contain positions from the second posting list only. We consider the expression tree for a given query Q. Each internal node in the tree contains pointers to its two children, the operator and a list of postings (documents and positions). Each leaf node corresponds to a query word and initially contains its posting list. For example, the leaf node that corresponds to the query word q_i initially contains the list of postings $L'(q_i)$. As in the example, the algorithm goes in two phases. In the first phase, the expression tree is evaluated in a bottom-up fashion starting from the root as shown with the following pseudo-code:

```
eval-bottom-up(node):
    If isLeafNode(node)
        node.list = $L'$(node.$q_i$);
    else
        node.list = node.op(eval-bottom-up(node.left),
                            eval-bottom-up(node.right));
```

where node.*op* is the operator of the current internal node. The bottom-up evaluation of the expression tree is analogous to the left-to-right evaluation in the example. In the second phase, the evaluation is top-down, where each node updates the lists of its children starting from the root as shown with the following pseudo-code:

```
eval-top-down(node):
    If isInternalNode(node)
        node.right.list = node.list;
        node.left.list = node.op-inv(node.left.list, node.right.list);
        eval-top-down(node.left);
        eval-top-down(node.right);
```

where node.*op-inv* is the inverse operator of the current internal node (in many cases the same operator). The top-down evaluation of the expression tree is analogous to the right-to-left evaluation in the example.

Lemma 9.3.2. *The above algorithm correctly computes the set of matching positions for each query word q_i in each of the documents from D in time $O(k \cdot \sum_{i=1}^{l} \log |L'(q_i)|)$.*

Proof. First we will show that the above algorithm works correctly. We consider the expression tree of a query Q. Observe that each node in the expression tree corresponds to a sub-query of Q. Let Q_u be the sub-query that corresponds to a node u and let Q_v and Q_w be the sub-queries that correspond to the left and the right child of u respectively (see Figure 9.2, left). Let $\mathrm{last}(Q_u)$ be the index of the last query word in Q, where $\mathrm{last}(q_i) = i$. Let $E(Q_u)$ be the result list of Q_u in form of an extended posting list (defined in the previous section) and let $L'(Q_u)$ be the result list of Q_u that contains only the postings of the last query word, i.e., $L'(Q_u) = \{p \in E(Q_u) \mid \mathrm{ind}(p) = \mathrm{last}(Q_u) \wedge \mathrm{doc}(p) \in D\}$, where $\mathrm{ind}(p)$ is the index of the query word in the extended posting list from which the posting p stems and $\mathrm{doc}(p)$ is the doc id of p. For each query word q_i, we have $L'(q_i) = \{p \in E(q_i) \mid \mathrm{doc}(p) \in D\}$, assuming that $E(q_i)$ is the resulting list of postings after applying the unary operator on q_i (if any). We would like to compute the list $L(Q_u) = \{p \in E(Q) \mid \mathrm{ind}(p) = \mathrm{last}(Q_u) \wedge \mathrm{doc}(p) \in D\}$ for each sub-query Q_u. Recall that each operator $\circ$ defines a binary relation over the set of all postings. Let R° be the relation defined by the operator $\circ$. Whenever $(p_1, p_2) \in R^\circ$, we will write $p_1 \circ p_2 = \top$, where $\top$ is the "true“ symbol. $L'(Q_v) \circ L'(Q_w)$ is then defined as $\{p_w \in L'(Q_w) \mid \exists p_v \in L'(Q_v), p_v \circ p_w = \top\}$, where $\circ$ is the query operator for Q_v and Q_w. Analogously, $L'(Q_v) \circ^{-1} L'(Q_w)$ is defined as $\{p_v \in L'(Q_v) \mid \exists p_w \in L'(Q_w), p_v \circ p_w = \top\}$. By using a simple induction and applying the definition of $L'(Q_v) \circ L'(Q_w)$ in the induction step, we obtain $L'(Q_u) = L'(Q_v) \circ L'(Q_w)$. This is what the first phase of our algorithm computes. Observe that $L(Q)$ is computed already at the end of the first phase of the algorithm since $L'(Q) = \{p \in E(Q) \mid \mathrm{ind}(p) = \mathrm{last}(Q) \wedge \mathrm{doc}(p) \in D\} = L(Q)$. Let l and r be the left and the right child respectively of the root of the expression tree, such that $L'(Q) = L'(Q_l) \circ L'(Q_r)$ (see Figure 9.2, right). Since $\mathrm{last}(Q) = \mathrm{last}(Q_r)$, it follows that $L(Q_r) = \{p \in E(Q) \mid \mathrm{ind}(p) = \mathrm{last}(Q_r) \wedge \mathrm{doc}(p) \in D\} = \{p \in E(Q) \mid \mathrm{ind}(p) = \mathrm{last}(Q) \wedge \mathrm{doc}(p) \in D\} = L(Q)$. Let the list $L''(Q_l)$ be defined as $L''(Q_l) = L'(Q_l) \circ^{-1} L'(Q_r)$ and let $p_l \in L''(Q_l)$. Since $p_l \in L'(Q_l)$ such that $\exists p_r \in L'(Q_r)$ with $p_l \circ p_r = \top$, the latter is equivalent to $p_l \in E(Q)$.

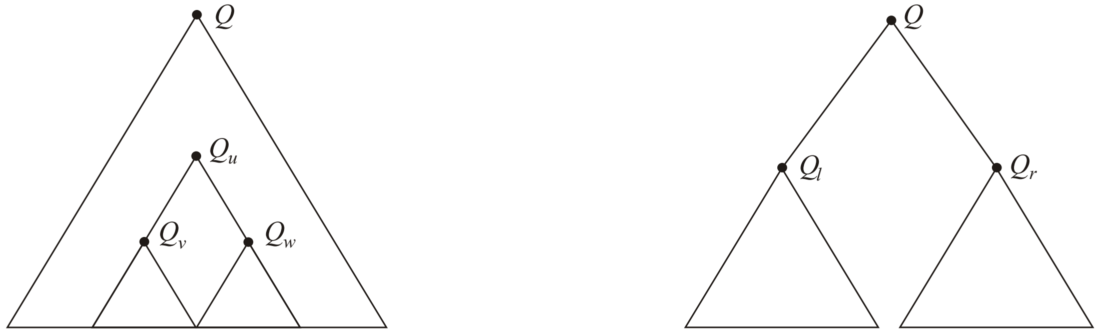


Figure 9.2.: Schematic representation of the expression tree for a query Q with subtrees that correspond to sub-queries.

Since $\text{ind}(p_l) = \text{last}(Q_l)$ and $\text{doc}(p_l) \in D$, by using the definition of $L(Q_l)$, the latter is equivalent to $p_l \in L(Q_l)$. Hence, $L''(Q_l) = L(Q_l)$. This is what the second phase of the algorithm computes in top-down fashion. The correctness can be shown by using induction on the depth in the expression tree. Namely, if we assume that all desired lists of nodes up to depth d in the tree are correctly computed, then by using the above argument we can correctly compute the lists of the nodes at depth $d + 1$ from their parents.

To compute the running time of our algorithm, assume for simplicity that each operator can be computed by an intersect-like operation. Two sorted lists of sizes x and y can be intersected in time $O(x \cdot \log y)$ (just binary search the elements from the first list in the second). Since $|D| = k$, the first phase can be completed in time $O(k \cdot \sum_{i=1}^{l} \log |L'(q_i)|)$. The second phase requires less time since the number of lists to be computed is equal to that from the first phase, however, their lengths are smaller. Without the above assumption, the total running time is dominated by the most expensive operator(s) in Q. □

Table 9.1 shows that our operator inversion approach adds less than 1% to the query processing time, and, unlike the extended posting lists, it does not require any modification of the internal posting list data structure.

9.4. Step I2: Computing the Snippet Positions

This section is about computing and ranking the actual snippets once the positions of each query word in each of the top-ranked documents have been computed. We start by showing how to compute the positions of all matching snippets given the positions of each query word and a partitioning of the document into segments. The produced snippets can be then ranked similarly as in the document-based snippet generation.

9.4.1. Algorithm for Computing the Snippet Positions

Computing the *positions* of the snippets (not the actual text, which is done in I3) to be output becomes an instance of the following simple problem, to be solved for each of the top-ranked documents.

(I2) Given l sorted lists of positions $L_1, \ldots, L_l$ pertaining to the l query words in a document, and a sorted list of positions $S = s_1, s_2, \ldots$ marking the segment beginnings in that document, compute a list of s tuples (seg_i, Pos_i), sorted by the seg_i, where seg_i is the index in S of the ith segment containing a position from one of the q_i, and Pos_i is the list of those positions from $L_1, \ldots, L_l$ which fall into that segment, together with the information from which of the lists each position has come. In simple terms, for each segment we would like to compute the set of positions from $L_1, \ldots, L_l$ (together with the corresponding query term indices).

Example 9.4.1. *Consider a 2-word query and a fixed document with 5 segments. Let* $L_1 = \{3, 8, 87\}$ *be the positions of the first query word,* $L_2 = \{13, 79\}$ *be the positions of the second query word, and* $S = \{1, 17, 43, 67, 98\}$ *be the first positions of each segment. Then the correct output of I2 would be the two tuples* $(1, (3, 1), (8, 1), (13, 2))$ *— segment 1 with three occurrences of the query words — and* $(4, (79, 2), (87, 1))$ *— segment 4 with two occurrences of the query words.*

Lemma 9.4.2. *For each hit displayed to the user, step (I2) can be executed in time* $O((|L_1| + \cdots + |L_l| + |S|) \cdot \log l)$.

Proof. (I2) can be implemented by a straightforward variant of an l-way merge, followed by a simple sort. Here we assume that a segment can be scored in linear time. □

Our experiments show that the computation involved in (I2) takes negligible time. This is understandable, since none of the three L_i or S or l are ever going to be very large, and we are only manipulating positions (numbers) in this step. The bulk of the time in the index-based approach is spent on disk seeks to access those parts of the documents that are used as snippets. For (I2), disk seeks are needed to fetch the list of segment boundaries.

9.4.2. Scoring of Snippets

Each tuple of positions computed by the algorithm in the previous section corresponds to a potential segment to be output. Now all that remains is to score the segments and return those with the m largest scores, for some user-defined m. In our experiments, we take $m = 3$ and score segments by a simple but effective scheme (described in Figure 2 of Turpin *et al.* [2007]). The scheme is based on a weighted sum of the number of query words in the segment, the number of distinct query words in the segment, the longest contiguous run of query words and few other similar quantities. Note that our index-based approach is compatible with any other scoring mechanism that works on the sentence level (or any other precomputed segmentation of the text). For example, many of the more complex approaches in the literature [Ko *et al.*, 2007; Varadarajan and Hristidis, 2006; White *et al.*, 2002; Goldstein *et al.*, 1999; Tombros and Sanderson, 1998] are naturally based on scoring individual sentences. Hence, the same (document-based) scoring schemes could be easily applied here with no difference in the quality of the produced snippets.[5] We note that a sophisticated snippet scoring scheme is not in the scope of this work.

9.5. Step I3: From snippet positions to snippet text

Given a list of positions — namely the positions of the top-ranked segments computed in (I2) — we want to obtain the actual text at these positions, which will then be displayed to the user. In this section, we show how to represent the documents so that this can be done efficiently. We begin by describing our block representation of documents that allows a trade-off between the amount of text scanned and the space overhead imposed by storing the document partitioning into segments.

We stress again that this final step in which the actual text gets generated is completely oblivious of the query words that have given rise to the position lists. As explained in Section 9.1.3, this has the valuable advantage that any kind of advanced search feature that defines what constitutes a match has to be implemented only once, on the query processing side.

9.5.1. Block Representation

In the document-based approach, the whole compressed document needs to be read from disk and scanned in order to determine which segments match the query. For large documents, this is an efficiency problem. As a heuristic, Turpin *et al.* [2007] suggest rearranging the documents such that segments that are likely to score high are at the beginning, and then consider only the head of each document.

For our approach, we could in principle read exactly the amount of text needed for the snippets and no more. There would be a price for this, however: first, we would (at least in theory) incur one disk seek per segment in large documents, and second, it would require us to store for every segment its offset in the document. To balance these conflicting goals we take the following approach.

Each document is divided into blocks, where each block covers a fixed number of p positions. We can easily identify the blocks containing the desired positions, given the output of step (I2). Every block is stored compressed on disk together with the relative offsets in the document. This minimizes the size of our document database, as well as the time to read blocks from disk. Smaller p would imply less text to be read and scanned, however, higher space overhead. Since our blocks are small, there is little value in encoding each token separately, as proposed in Turpin *et al.* [2007]. Instead, we compress each block as a whole by using zlib (with

[5] Note that this might sometimes require performing the segment scoring in step (I3) where the actual segment text is produced instead of doing it in step (I2) which manipulates with positions only.

the default compression level 6). As shown in Section 9.7.2, this gives us slightly better compression than the compression achieved in Turpin *et al.* [2007].

To achieve a higher compression ratio, instead of zlib one could use a compression method based on the Lempel-Ziv (LZ) family that take into advantage global repetitive properties of the document collection. Hoobin *et al.* [2011] proposes a compression scheme for fast random access that is an alternative to the blocked representation. This method is based on a string data structure called relative Lempel-Ziv factorization and allows much higher compression ratios, yet faster document retrieval. We would like to note that novel compression schemes for fast document retrieval are not in the scope of this work.

9.5.2. Compression of the Blocks

To minimize the total size of our document database, as well as the time to read the blocks from disk, we store the blocks in compressed format. Since our blocks are small, there is little value in encoding each token separately, as proposed in Turpin *et al.* [2007]. Instead, we compress each block as a whole by using zlib (with the default compression level 6). As shown in Section 9.7.2, this gives us slightly better compression than the compression achieved in Turpin *et al.* [2007].

9.5.3. Snippet Text and Highlighting Matches

After each of the relevant blocks is read and decompressed, the text corresponding to the given segment positions is easily obtained by a simple scan over these blocks. The actual highlighting is done at the end of (I3) - when the snippet text has been fully reproduced. In order to use the same word boundary definition as the parser, each indexed word is marked with a special character produced at index time. This provides the exact information for every word position and permits highlighting of multi-word matches (e.g. `Tony Blair` matching `politician` in Figure 9.1). Note that the tuples corresponding to segment positions computed in (I2) contain the information whether a certain position is matching.

9.6. Snippet Caching

This section is about snippet caching. As we will see in Section 9.7, the times for steps (I2) and (I3) are dominated by the disk seeks, i.e., the time required to seek the list of the first positions of the segments, and the time required to seek the blocks to be read from disk. A similar observation has been made in Turpin *et al.* [2007] for the document-based approach. They suggest two caching strategies, one static and one dynamic, with the goal of avoiding as many disk seeks as possible. Both of these strategies cache whole documents because, as discussed earlier, finding the matching segments requires a scan over the whole document.

In this section, we start by describing our segment caching algorithm. We then compare it against document caching and show that under certain assumptions, snippet caching achieves substantially higher cache hit-ratio when the cache sizes are equal. We confirm this experimentally in Section 9.7.4.

9.6.1. Segment Cache

Our approach naturally allows caching of *individual segments* instead of whole documents. This is because (I2) outputs the indices of segments (which were obtained from the inverted index) to be output before we ever look at the actual documents. When a segment with index j from a document with id i is requested for the first time, it is fetched from disk, and stored in memory under the key (i, j) using a hash map. Whenever the segment with key (i, j) is requested again, we fetch it directly from memory. When a pre-specified amount of memory (the capacity of the cache) is exceeded, we evict the least recently used (LRU) segment(s) from the cache until there is enough space again.

The segment cache has two main advantages over the document cache. First, it allows higher throughput since involves copying significantly less amount of text to memory. Second, and even more importantly, it makes much better use of the available memory than whole-document caching. In the following, we provide theoretical as well as experimental evidence to support this claim.

Note that storing only selected segments would be of no use for the document-based approach, because for a new query - document pair we do not know which segments to retrieve before we scan the entire document.

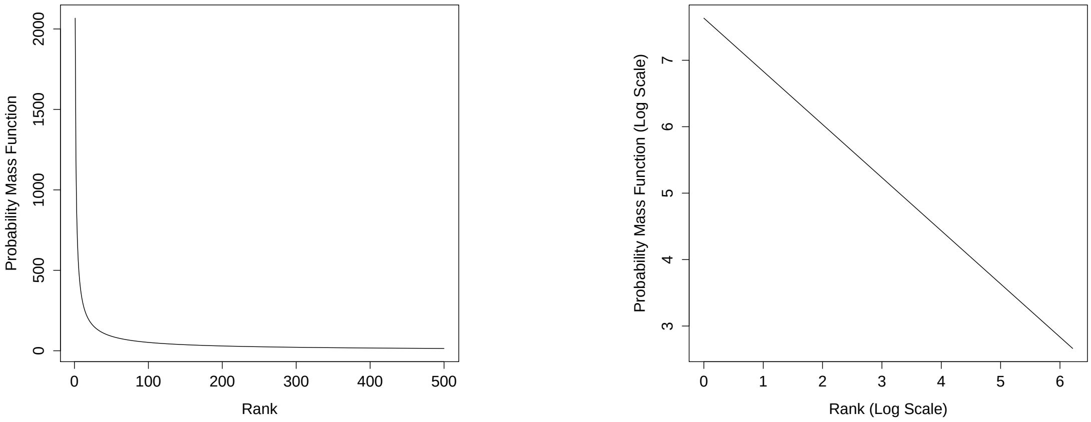


Figure 9.3.: Ordinary (left) and log-log plots (right) of the probability mass function of a Zipfian distribution with $\alpha = 0.8$ and $n = 500$.

Turpin *et al.* [2007] and Tsegay *et al.* [2009] address the problem of storing whole documents in the cache by moving segments with likely high scores to the beginning of the documents. This, however, risks an occasional loss of relevant segments, which may not be tolerable in some applications.

9.6.2. Distribution of Document and Segment Requests

In this section, we provide experimental evidence to show that the top segment requests exhibit distribution similar to that of the document requests.

Distribution of Document Requests

Previous studies suggest that document requests follow an approximate *Zipfian distribution*, also known as *Zipf's law* [Almeida *et al.*, 1996; Breslau *et al.*, 1999]. Zipf's law states that the probability of a request for the i'th most frequently seen document is equal to $H_{n,\alpha}^{-1}/i^{\alpha}$, where $\alpha > 0$ is a constant, n is the total number of documents and $H_{n,\alpha}$ is the generalized harmonic number of order n of α defined as $\sum_{i=1} 1/i^{\alpha}$. For a fixed n, a larger α means a higher locality of reference of the document requests. A large n, on the other hand, means a lower locality of reference of the document requests. This is because the probability mass is spread among a larger number of documents.

The page request distribution seen by web caches is investigated by Breslau *et al.* [1999] by using traces from variety of sources. According to their study, typical values of α range in the interval 0.7 up to 0.8 (i.e., we consider $0 < \alpha < 1$). Figure 9.3 shows that the plot of the Zipfian distribution has a long tail and it is represented by a straight line (with a slope equal to $-\alpha$) when both axes are in log scale.

Distribution of Segment Requests

Figure 9.4 shows a log-log plot of the distribution of document and segment request for the TREC Terabyte collection. It is clear that the plots have similar slopes. Figure 9.5 shows the segment request frequency of the segments requested in the top-100 documents on two of our test collections. For example, the frequency of the i-th ranked segment is calculated as an average frequency of all segments with rank i (segments with average frequency of less than 1 are not shown). The left side of Figure 9.5 shows that, unlike the Zipfian distribution, the distribution of segment requests does not have a long tail. In fact, most of the requests are concentrated on a single segment per document. More precisely, 85% of all segment requests on the DBLP collection and 78% of all segment requests on the TREC Terabyte collection are requests for a single segment per document. The plots in log-log scale show that the distribution of segment requests is more skewed than the Zipfian distribution. Figure 9.6 shows the fraction of top-10 segment requests from all segment request in the top-100 documents on the DBLP collection. In general, 95% of all segment requests are concentrated on the 10 most frequently requested segments per document.

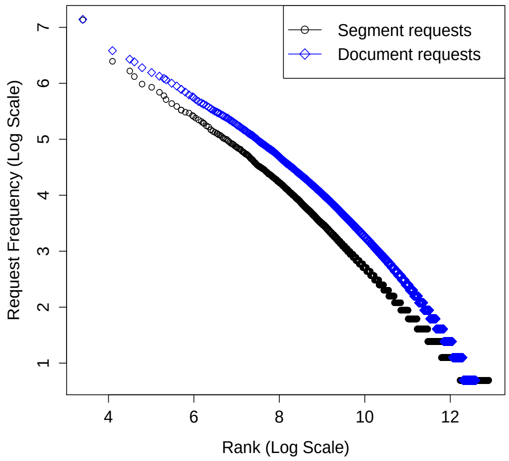


Figure 9.4.: A log-log plot of the frequency of document and segment requests versus document and segment ranking on the TREC Terabyte collection (see Section 9.7.1).

Ceccarelli *et al.* [2011] try to answer a similar question, namely, how many different snippets have to be generated for a single document. They come to a similar conclusion as we do, namely, that the number of different snippets for a given document is small. In particular, according to their experimental analysis, 99.96% of all documents have less than 10 snippets associated with them, and about 92.5% have only a single snippet. In this work, instead of caching whole documents, the authors construct a so called *super-snippet* that contain the most frequently requested snippets of that document. It is not hard to see that our fine-granularity cache does this naturally and transparently by caching only the relevant snippets. After a warm-up period, our segment cache will automatically contain only the snippets with high likelihood to be requested again in the future.

9.6.3. Segment vs Document Cache

In this section, we compare the asymptotic cache hit-ratios (the hit-ratio for infinite number of document requests) and the convergence rates of the document and the segment cache by using the observations from the previous section.

First we would like to point out that given the fact that only a small number of snippets are generated from each document, it is intuitive that the snippet cache will make a better use of the available memory than the document cache, and hence, achieve a higher cache hit-ratio. To show this more formally, we consider a finite cache with a capacity of m documents that receives a stream of independent requests. In addition, we make a number of simplifying assumptions. First, we assume a Perfect-LFU removal policy, i.e., a cache that always contains the m most frequently requested documents (segments). Second, we assume that all documents have the same size, and third, we assume that only the top ranked segment is cached and shown to the user. The asymptotic hit-ratio of the Perfect-LFU replacement policy (see Breslau *et al.* [1999]) is then given by

$$\sum_{i=1}^{m} \frac{1}{H_{n,\alpha}} \cdot \frac{1}{i^{\alpha}} = \frac{H_{m,\alpha}}{H_{n,\alpha}} \tag{9.1}$$

Let $c > 1$ be the average number of segments per document. Our segment cache could then fit roughly $c \cdot m$ out of $c \cdot n$ segments. Let $0 < f \leq 1$ be the fraction of all segments that contribute to the overall distribution of segment requests, i.e., we assume that in average $f \cdot c$ relevant segments are generated from each document. Let α_D and α_S be the parameters of the Zipfians that correspond to the document and the segment requests respectively. It is not hard to see that $\alpha_D \geq \alpha_S$ since each segment request is preceded by a request of the document that contains that segment. However, from the discussion in the previous section, we will assume that $\alpha_D \approx \alpha_S$. Under the above assumptions, it is not hard to show the following:

Lemma 9.6.1. *(i) The segment cache has higher asymptotic hit-ratio than the document cache; (ii) The document cache converges faster than the segment cache.*

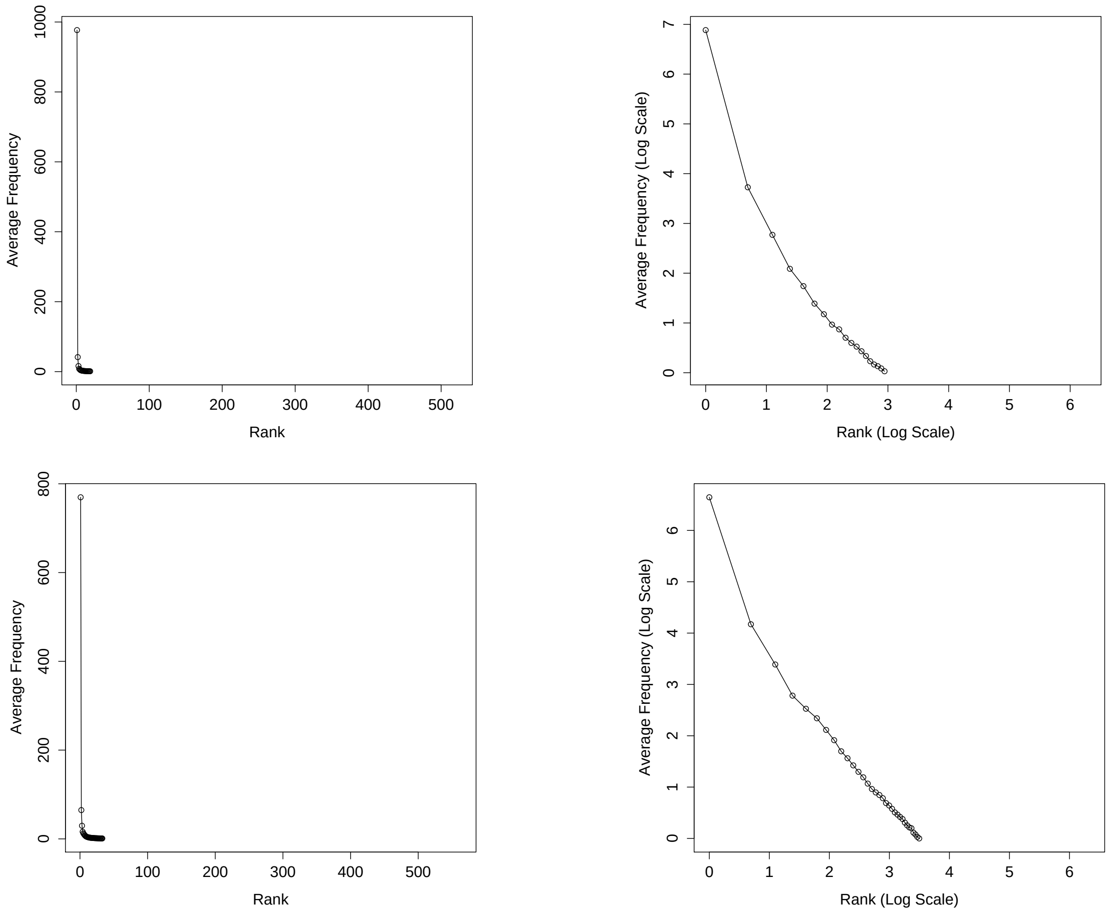


Figure 9.5.: Average frequency of segment requests in a document, shown in ordinary (left) and log-log scale (right). The frequencies have been calculated as averages over the 100 most frequently requested documents from the TREC Terabyte (above) and the DBLP collection (below) (see Section 9.7.1).

Proof. To show (i), we consider the asymptotic hit-ratio of the segment cache given by

$$h(c, f) = \frac{H_{c \cdot m, \alpha}}{H_{f \cdot c \cdot n, \alpha}} \tag{9.2}$$

as a function of c and f, where $h(1, 1)$ is equal to the asymptotic hit-ratio of the document cache given in Equation 9.1. By approximating $H_{n,\alpha}$ by the integral $\int_1^n 1/x^\alpha \, dx$, it is a matter of simple calculus to show that $h(c, f)$ is monotically increasing function of $c > 1$ and hence $h(c, f) > h(1, 1)$. Figure 9.8 compares the asymptotic hit-ratios of the segment and the document cache for different values of m and f.

To show (ii), we make use of the following simplification from Breslau *et al.* [1999]. We consider a cache of infinite size so that all previously requested pages remain in the cache. We assume that R documents have been already requested and consider the cache hit-ratio at the $R + 1$'st request. Let the $R + 1$'st request be a request for the document with rank i. The probability for a cache hit is then equal to the probability that this document has been already requested, which in turn is equal to

$$1 - \left(1 - \frac{H_{n,\alpha}^{-1}}{i^\alpha}\right)^R$$

Consider the event that the $R + 1$'st document request causes a cache hit. By the law of total probability, the

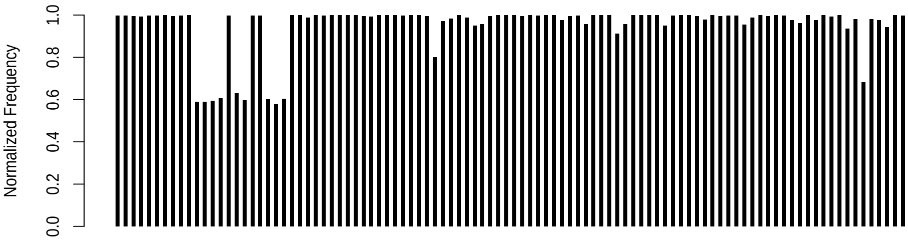


Figure 9.6.: The fraction of segment requests which belong to the 10 most frequently requested segments in a document shown for the 100 most frequently requested documents from the DBLP collection. Each bar corresponds to a single document.

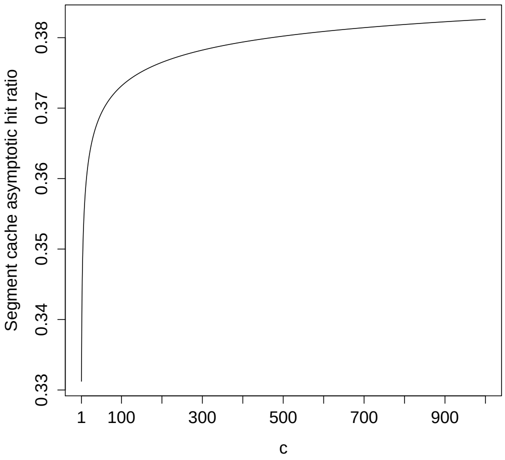


Figure 9.7.: Asymptotic hit-ratio of the segment cache ($h(c, f)$) for $f = 1$ and increasing c ($\alpha = 0.8; m = 1{,}000; n = 100{,}000$).

probability of this event is equal to

$$p(R,n) = \sum_{i=1}^{n} \frac{H_{n,\alpha}^{-1}}{i^{\alpha}} \left(1 - \left(1 - \frac{H_{n,\alpha}^{-1}}{i^{\alpha}}\right)^{R}\right) \tag{9.3}$$

Analogously, the probability for a cache hit at the $R + 1$'st segment request is equal to $p(R, f \cdot c \cdot n)$. We will show that for any fixed R, $p(R, n)$ is a monotonically decreasing function of n, which would imply that $p(R, f \cdot c \cdot n) \leq p(R, n)$. By dropping i^{α} from the second term in Equation 9.3, we obtain

$$\begin{aligned} & p(R,n) \\ \leq & \left(1 - \left(1 - \frac{1}{H_{n,\alpha}}\right)^{R}\right) \cdot \sum_{i=1}^{n} \frac{H_{n,\alpha}^{-1}}{i^{\alpha}} \\ = & \ 1 - \left(1 - \frac{1}{H_{n,\alpha}}\right)^{R} \end{aligned}$$

This expression is a monotonically decreasing function of n since $H_{n,\alpha}$ is a monotonically decreasing function of n. Hence, $p(R, n)$ must be a monotonically decreasing function of n. □

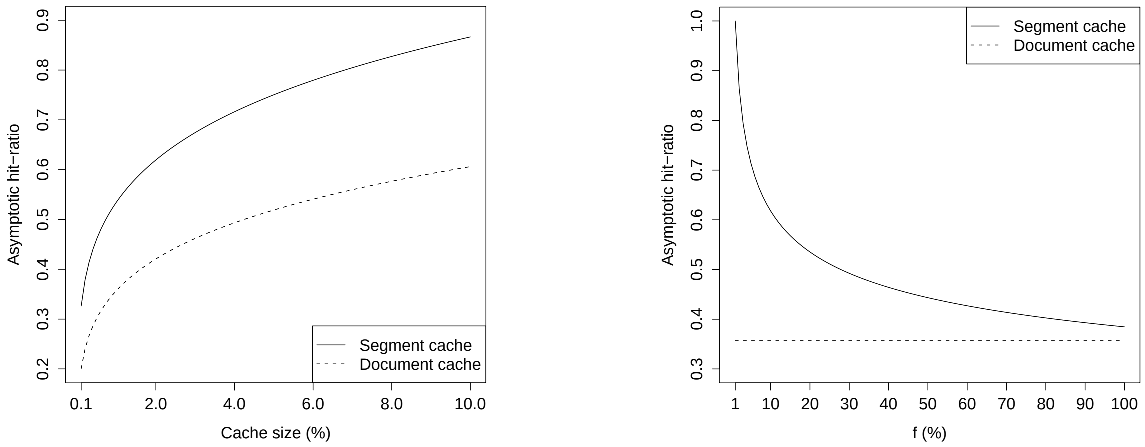


Figure 9.8.: Asymptotic hit-ratio of the segment and the document cache for $n = 1,000,000$ (1-million document collection) when the cache size varies (left); and when f (the percentage of relevant segments) varies and $m = 10,000$ documents (right).

Table 9.2.: Size of the data with all tags, mark-up and binary data (e.g. pdf) removed, size of the index (used for query processing), and size of the document databases (for snippet generation) for our implementation of the index-based approach (fourth column), and for Turpin et al's implementation of the document-based approach (fifth column), on all three test collections.

Collection	Raw Data	Index	Docs DB	Turpin
DBLP	8.7 GB	0.47 GB	0.35 GB	0.42 GB
Wikipedia	12.7 GB	4.52 GB	3.41 GB	3.53 GB
GOV2	132.9 GB	38.2 GB	36.3 GB	45.8 GB

9.7. Experiments

We ran various experiments to evaluate our implementation of index-based snippet generation, and compared it against the document-based approach as well as against the query algebra approach as implemented by the Wumpus search engine; see Section 9.2.

All our experiments were run on a machine with two dual-core AMD Opteron processors (2.8 GHz and 1 MB cache each), 16 GB of main memory, running Linux 2.6.20 in 32-bit mode. All our code is single-threaded, that is, each run used only one core at a time. All our code is in C++ and was compiled with g++ version 4.1.2, with option -O3. Before each run of an experiment, we repeatedly read a 16 GB file (that was otherwise unrelated to our experiments) from disk, until read times suggested that it was entirely cached in main memory (and thus all previous contents flushed from there).

9.7.1. Collections and Queries

DBLP: 31,211 computer science articles listed in DBLP, with full text and meta data, and 20,000 queries extracted from a real query log, for example `matrix..factor`. The index was built with support for fuzzy and prefix search, as explained in Section 9.1.1. All queries were run as proximity queries, as in the example of Figure 9.1, where `..` means that the query words must occur within five words of each other.

This collection was chosen to test our approach on fuzzy and prefix search as well as boolean and proximity search.

For the cache efficiency experiment on this collection we used another query log of roughly 150,000 boolean queries.

Semantic Wikipedia: 2,843,056 articles from the English Wikipedia with an integrated ontology, as described in Bast *et al.* [2007] and 500 semantic queries from Bast *et al.* [2007], for example `finland city`, where `city`

Table 9.3.: Breakdown of the elapsed snippet generation times of our implementation of the index-based approach by steps I1, I2, and I3. Step I0 is the ordinary query processing, which provides the ids of the top-ranked documents. All entries are averages per query, that is, the times in the last three columns relate to the generation of 10 snippets at a time.

Collection	I0	I1	I2	I3
DBLP	45 ms	0.2 ms	6.0 ms	13.7 ms
Wikipedia	113 ms	0.1 ms	51.6 ms	8.7 ms
GOV2	144 ms	0.9 ms	37.5 ms	12.2 ms

Table 9.4.: Elapsed snippet generation times (without using caching) of our implementation of the index-based approach compared to Turpin et al's implementation of the document-based approach and to the GCL query algebra approach implemented in the Wumpus search engine on the TREC GOV2 collection.

Ours	Turpin	Wumpus
46.6 ms	49.5 ms	396.7 ms

does not only match the literal word but also all instances of that class. To compute these matches, information from several documents needs to be *joined*; for details, we refer to Bast *et al.* [2007].

This collection was chosen to test our approach for queries that require a join of information from two or more documents for each hit. As explained in Section 9.1.1, the document-based approach would not be able to produce query-dependent snippets in this case, because the hit document does not contain all the required information to assess which words or phrases did actually match the query.

TREC GOV2: 25,204,103 documents from the TREC GOV2 corpus[6], and 420,316 queries from the TREC 2006 efficiency track and all queries from the AOL query log which lead to at least one hit. An example query is `first landing moon`.

This collection was chosen to test the efficiency of our approach on a very large collection and for a large number of queries. Note that Turpin et al. experimented with the somewhat smaller wt100g collection (102 GB of raw data) combined with the somewhat larger Excite query log (535,276 queries).

9.7.2. Space Consumption

We built the document database as described in Section 9.5, with each block covering 1000 positions, which amounts to an average of around 4 KB of compressed text. This gave the best snippet generation times overall.

Table 9.2 shows that the size of our document database is roughly equal to the size of the index used for query processing (which is of a different kind for each of the collections, see Section 9.7.1).

Our document database is slightly smaller than that described in Turpin *et al.* [2007]. This is because in our index-based approach every block is compressed as a whole (using zlib), which compresses the original text[7] to about 27%. The id-wise compression of Turpin *et al.* [2007], which we briefly described in Section 9.2, achieves a compression ratio of only about 35%. A part of our document database is auxiliary data needed to locate the document data within the single big file, as well as information on the segment bounds and the block offsets of each document (as before, for our experiments we consider a simple segmentation into sentences). This auxiliary data takes about 5% of the total size of the document database for each of our three collections.

For comparison, the RLZ compression scheme from Hoobin *et al.* [2011] mentioned in Section 9.5.1, achieves a compression ratio of 9-16% and simultaneously somewhat faster random segment access.

Table 9.5.: Cache hit-ratios for the two caching strategies and four different cache capacities on the TREC GOV2 and the DBLP collection. The numbers in parentheses give the total size of the cached items after all queries have been processed.

Collection	Method	128 MB	256 MB	512 MB	1024 MB
GOV2	Segment cache	0.69 (128 MB)	0.74 (196 MB)	0.74 (196 MB)	0.74 (196 MB)
	Document cache	0.20 (128 MB)	0.28 (256 MB)	0.40 (512 MB)	0.56 (1024 MB)
DBLP	Segment cache	0.87 (40 MB)	0.87 (40 MB)	0.87 (40 MB)	0.87 (40 MB)
	Document cache	0.53 (128 MB)	0.64 (256 MB)	0.82 (512 MB)	1.00 (1024 MB)

9.7.3. Snippet Generation Time

We ran our index-based snippet generation implementation on all three test collections, for the queries described in Section 9.7.1. On the GOV2 collection, we compared it to the document-based implementation as well as to the GCL query algebra approach implemented in the Wumpus search engine. All experiments were carried out by computing the snippets of the top-10 retrieved documents.

The main observation from Table 9.3 is that the operator inversion from step (I1), which provides the basis for all other steps in the index-based approach, takes negligible time compared to all other steps. The bulk of the time spent for (I2) and (I3) is disk seek time, that is, the cost of our snippet generation is essentially the cost of the required disk seeks. Turpin *et al.* [2007] have come to the same conclusion for their document-based approach.

In fact, the table shows that, unlike query processing times, snippet generations times do not depend on the size and nature of the collection, but rather on the number of queries. We also measured a slight dependency on the average document length (which is largest for DBLP: 32 kB, and smallest for GOV2: 10 kB), which influences the length of the list of segment bounds read in step (I2).

Table 9.4 shows that for ordinary keyword queries our implementation of the index-based approach is slightly faster than the document-based approach. This is because the index-based approach does not have to read the whole document from disk, but rather only those blocks containing the positions computed in steps I1 and I2. Note, however, that apart from caching, there is no way to be much faster for basic keyword search, but that the document-based approach simply cannot be used as is for more complex queries like those for DBLP and Wikipedia.

Worst-case snippet-generation times are much higher for the document-based approach, for example, 1,453 of the 420,316 queries on GOV2 required more than 1 second (due to very large documents being involved), whereas for our index-based approach snippet generation never exceeded 100 milliseconds.

The snippet generation times for Wumpus were significantly higher than that of our implementation of the index-based approach. As explained in Section 9.2, this is because the underlying algorithm touches *every* posting of each query word, which is exactly what our operator inversion avoids. Put simply, one could thus say that we achieve the power of the index-based approach (also realized in Wumpus) at the computational cost that can be achieved by the (less powerful) document-based approach.

9.7.4. Caching

As explained in Section 9.6, the index-based approach allows caching on the level of individual segments, instead of on the level of whole documents as is the case for the document-based approach. To evaluate the potential of this finer granularity caching, we carried out experiments on both caching methods by using various cache sizes on the (relatively large) GOV2 and the (relatively small) DBLP collection. For each method, we used the first half of the queries to fill the cache, and then measured the hit-ratio for the second half of the queries. The cache eviction strategy was LRU, as explained in Section 9.6.

Table 9.5 shows that the fine-grained segment caching results in significantly better cache hit-ratio than the coarse document caching, as predicted in Section 9.6.3. This is especially pronounced for small cache sizes for two reasons: first, our model predicts that the gap between the cache hit-ratios of the two caching methods

[6] http://www-nlpir.nist.gov/projects/terabyte

[7] We measure compression ratio with respect to the plain text, with all mark-up and tags removed. The compression ratio in Turpin *et al.* [2007] has been measured with respect to the size of the original html, pdf, etc. files.

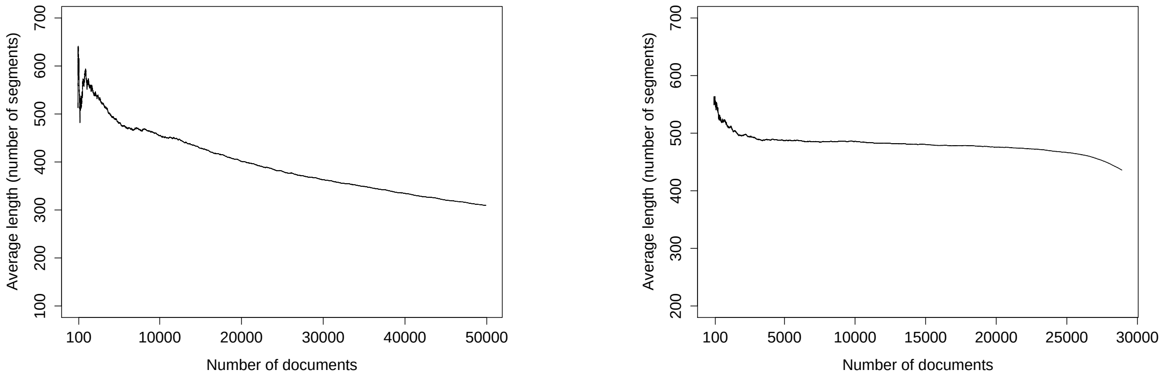


Figure 9.9.: Average document size of the first n most frequently accessed documents given in number of segments (sentences) on the TREC GOV2 collection (left) and the Wikipedia collection (right).

becomes larger when the size of the cache is small; and second, on larger cache sizes the top-segments cache requires a longer request stream than the whole-document cache to converge and reach its asymptotic hit-ratio, also predicted by our model. The number of queries in our case was simply too small to make full use of the segment cache with larger cache sizes. Namely, only 196 MB were sufficient to store the snippets returned for all top-10 hits for all of our 420,316 queries on the TREC GOV2. For larger query logs we would expect the cache hit-ratio of the segment cache to go up further.

Document ranking has an additional negative effect on document caching. To see this, recall that one of the assumptions in our model is that all documents have roughly the same size. However, Figure 9.9 shows that more frequently accessed documents are more likely to have larger sizes than less frequently accessed documents. This effect is especially pronounced on the TREC GOV2 collection[8]. As a result, the document cache fills up more quickly. Note that this is in accordance with the scope hypothesis in information retrieval stating that the likelihood of document relevance increases with document size [Losada *et al.*, 2008]. For the segment cache the distribution of document sizes is not an issue since the segment sizes are independent of the document sizes.

[8] We used a standard BM25 ranking function

10. Conclusions and Future Work

10.1. Recap of Main Contributions

In this thesis, we have addressed two variants of the fuzzy matching and the fuzzy search problems in large text collection. The first variant deals with *keyword-based* search while the second variant deals with *prefix* or *autocompletion* search. Our algorithms are significantly more efficient and scale to larger collections than previous methods. They also permit *query suggestions* based on the contents of the document collection rather than a pre-compiled list, as in most previous work. All of our algorithms have been integrated into a fully-fledged fuzzy search system within the CompleteSearch engine.

In the first part of the thesis, we have formalized and given solutions to the problems that are central to our fuzzy search. We have presented two new practical algorithms for the fuzzy word matching problem with two different trade-offs (Chapter 3). Depending on the threshold, our algorithms improve the previously best algorithms for up to one order of magnitude. We have then shown how to extend each of the aforementioned algorithms into efficient fuzzy prefix matching algorithms (Chapter 4). Similarly, we have achieved improvement of up to one order of magnitude over the previously best algorithms. For the fuzzy keyword-based and fuzzy prefix search problem, we have proposed two novel data structures, called fuzzy-word index and fuzzy-prefix index together with a new query processing algorithm (Chapter 5). If the respective indexes reside in memory, we achieve an improvement of up to a factor of 9 for fuzzy prefix search and up to a factor of 12 for fuzzy keyword-based search. We have also proposed an algorithm for computing a ranked list of query suggestions, including an algorithm for computing phrase completions (Chapter 6). Our algorithms do not require query logs and take only a small fraction of the query processing time. Finally, we have shown how to incorporate additional practically relevant extensions into our fuzzy search (Chapter 7).

In the second part of the thesis, we have addressed two important engineering aspects of a fuzzy search system. The contribution in these works, however, extends beyond the scope of fuzzy search. First, we have considered the problem of efficient construction of the HYB index for both, exact search as well as fuzzy search (Chapter 8). We have shown that for fuzzy search our index can be constructed as fast as an ordinary inverted index and up to twice as fast for exact search. Second, we have addressed the problem of generating textual snippets for complex queries and non-literal matches, such as those produced by fuzzy search (Chapter 9). Besides the fuzzy search operator, our algorithm supports other advanced query operators both efficiently and without code duplication required by the standard approach.

10.2. Loose Ends and Future Research Directions

In this section, we discuss the issues as well as the questions that have been left open in this thesis. In addition, we point out a few interesting future research directions that have arisen along the way.

10.2.1. Avoiding Reading Large Amounts of Data from Disk

Recall that whenever a new query is submitted from scratch (pasted), our approach requires reading *each* of the respective fuzzy inverted lists. As already mentioned in Section 5.8.2, any approach with this requirement would be problematic in a scenario where the user has *pasted* a long query and the indexed corpus is *large* and resides on *disk*. This is because the fuzzy inverted lists are typically long, resulting in very large amount of data to be read from disk. As already suggested by our experiments, this imposes a bottleneck and calls for approaches without this requirement. One such algorithm is `ForwardList` from Ji *et al.* [2009] (described in Section 5.1). However, this algorithm requires reading the *forward list* of each document from the shortest fuzzy inverted list. If M is the average number of distinct terms per document and if the lengths of the fuzzy inverted lists do not vary significantly, then this is equivalent to simultaneously reading M fuzzy inverted lists from disk. Hence, an interesting open question is if such an approach exists or if fuzzy search for at least our (strict) definition inevitably involves dealing with large amounts of data.

10.2.2. Top-k Query Processing

An obvious idea for further improving the efficiency and scalability of fuzzy search would be to combine our approach with *top-k techniques* in order to avoid *fully* materializing the lists of all matching postings. This is especially relevant for *large* collections with abundance of hits for a popular query. Such an approach was already discussed in Chapter 7, however, experimental evaluation for its practical effectiveness was left out. Still, even if it turns out to be promising in practice, a weakness of this approach is that computing the result hits *incrementally* as the user types the query would not be so easy as the top hits of the new query would not necessarily be a subset of the top hits of the old query (especially when the user starts a new keyword). Another raised question is whether the top-k hits are enough to compute good query suggestions.

10.2.3. Parallel Processing

Parallel or distributed processing is another obvious way to scale up our fuzzy search on very large collections. The two standard partitioning approaches from Moffat *et al.* [2006], namely, the *term-based* approach, where each machine becomes responsible for a subset of words, and the *document-based* approach, where each machine becomes responsible for a subset of documents, could, in principle, be employed in our setting without major obstacles. Parallelization of our fuzzy search can be employed on a finer grain, too. Parallelizing a fuzzy word or prefix matching algorithm is straightforward: split the dictionary into several equally sized parts which are then independently indexed and queried. The final result is obtained by simply combining the results from each part. Recall that our query processing typically involves intersecting one with few other lists and then merging the obtained results. Each of these intersections could be computed *independently* on a different core. Since the number of resulting lists is small, they could be either merged in pairs on different cores or merged one by one using *parallel merging*. However, it is unlikely that it is worth investing multiple cores for a rather modest total speed up compared to using the available cores for simultaneously serving as many users as possible.

10.2.4. Evaluating the Search Quality

Although a variant of our fuzzy search is already deployed in searching the DBLP bibliography[1], the quality of the provided hits as well as the quality of the query suggestions have not been experimentally evaluated but rather taken for granted. This is because, first, the focus of this thesis has been predominantly on *efficient* realization of these features, and second, in *vertical* or *domain-specific search*, where queries with small hit sets are common, *recall* is as important as precision and sometimes even more important. However, it would be interesting to conduct a user study about the *usefulness* of our system and the *quality* of the delivered results on large text collections that require “non-trivial“ ranking. It would be also interesting to compare the quality of our query suggestions to a system where they are computed from a past user input, i.e., query logs.

[1] The DBLP computer science bibliography is daily search by hundreds of users at http://www.dblp.org/search

A. Appendix

A.1. Proofs Omitted for Brevity

Proof of Lemma 3.2.3. Since $\mathrm{WLD}(w_1, w_2) \leq \delta$, there exist a sequence of edit operations (insertions, deletions, substitution) $(O_1, \ldots, O_T)$ that transforms w_1 into w_2 as well as corresponding sequence of edit operations $(O'_1, \ldots, O'_T)$ that transforms w_2 to w_1. Let $pos(O_i)$ be the position of O_i. We will construct l-tuples of delete positions p_1 and p_2 in w_1 and w_2 in δ steps such that $s(w_1, p_1) = s(w_2, p_2)$ as follows. If O_i is a substitution, then there exist a corresponding substitution O'_i on w_2. We delete the character with position $pos(O_i)$ from w_1 and the character with position $pos(O'_i)$ from w_2 and include $pos(O_1)$ and $pos(O_2)$ to p_1 and p_2 respectively. If O_i is an insertion, then there exists the corresponding delete operation O'_i on w_2. We apply O'_i on w_2 and append $pos(O'_i)$ to p_2. If O_i is a deletion, we apply O_i on w_1 and append $pos(O_i)$ to p_1. Observe that at each step the distance between the resulting strings must decrease by 1. Hence, after δ steps the resulting strings must be equal. Since at each steps the length of the strings decreases by at most 1, the final string has length at least $\max\{|w_1|, |w_2|\} - \delta$. □

Proof of Lemma 3.2.4. Assume $\mathrm{WLD}(w_1[k], w_2[k]) > \delta$. For the sake of clarity we will assume that all δ errors in w_1 and w_2 have taken place on positions at most k. An alignment between w_1 and w_2 is defined by partitioning w_1 and w_2 into the same number of possibly empty substrings $p_1 p_2 \ldots p_l$ and $s_1 s_2 \ldots s_l$ such that $p_i \rightarrow s_i$ (or equivalently $s_i \rightarrow p_i$) with cost $c(p_i, s_i)$ (p_i and s_i cannot be empty in the same time). WLD computes an alignment between w_1 and w_2 with minimal cost. Since $\sum_{i=1}^{j} c(p_i, s_i) \leq \delta$ for any $1 \leq j \leq l$, $\mathrm{WLD}(w_1[k], w_2[k]) > \delta$ implies that $w_1[k]$ and $w_2[k]$ cannot be represented as $p_1 \ldots p_i$ and $s_1 \ldots s_i$ for some i, i.e., p_i and s_i are not fully contained in $w_1[k]$ and $w_2[k]$ respectively. Without loss of generality we can assume that p_i contains characters that match characters in the suffix of w_2. Hence, we can restore the alignment by removing $t \leq \delta$ characters from p_i obtaining $\mathrm{WLD}(w_1[k-t], w_2[k]) \leq \delta$. This means there is a transformation $s_t \rightarrow p_t$, $t < i$ such that $p_t = \epsilon$ and $|s_t| = t$. Let w'_2 be the resulting string after applying $s_t \rightarrow p_t$ to $w_2[k]$. We now have $\mathrm{WLD}(w_1[k-t], w'_2) \leq \delta - t$. According to Lemma 3.2.3 we can find a matching subsequence between $w_1[k-t]$ and w'_2 by applying at most $\delta - t$ deletions. □

Proof of Lemma 5.3.4 (second part). Let C be a given clustering and let $Ov(C)$ be its index space overhead as given in Definition 5.3.2:

$$Ov(C) = \frac{\sum_{w \in W} \mathrm{tf}_w \cdot c_w}{\sum_{w \in W} \mathrm{tf}_w}$$

Observe that due to the Zipf's law, $Ov(C)$ is mainly determined by the c_w's for which $\mathrm{tf}_w > t$. Assume that the number of frequent words is f and that the term frequency of the word with the ith rank w_i is given by

$$\mathrm{tf}_{w_i} = N \cdot \frac{1}{c \cdot i^{\alpha}}$$

where $c = \sum_{i=1}^{f} 1/i^{\alpha}$ is the normalization factor of the Zipfian distribution and $N = \sum_{\mathrm{tf}_w \geq t} \mathrm{tf}_w$. Then we obtain

$$\begin{aligned} Ov(C) &\approx \frac{1}{N} \cdot \sum_{\mathrm{tf}_w \geq t} \mathrm{tf}_w \cdot c_w \\ &= \frac{1}{N} \cdot \sum_{i=1}^{f} N \cdot \frac{1}{c \cdot i^{\alpha}} \cdot c_{w_i} \\ &= \frac{1}{c} \cdot \sum_{i=1}^{f} \frac{1}{i^{\alpha}} \cdot c_{w_i} \end{aligned}$$

If we assume that the c_{w_i}'s are equally distributed, due to linearity of expectation we obtain

$$\mathbb{E}[Ov(C)] \approx \frac{1}{c} \cdot \sum_{i=1}^{f} \frac{1}{i^{\alpha}} \cdot \mathbb{E}[c_{w_i}] = \mathbb{E}[c_{w_i}]$$

According to Definition 5.3.3, the latter is approximately equal to $\overline{SF}$. □

Proof of Lemma 5.4.3. Assume that each $*$-prefix contains s “don't care“ characters. For each combination of errors in q (deletions, insertions, substitutions) we show how to find a family of $*$-prefixes p in W_k^{**} with $\text{PLD}(q, p) \leq \delta$ that cover S_q (note that the “don't care“ characters do not match any other characters when computing the PLD between two prefixes).
(i) *insertion errors:* for simplicity, assume that $\delta = 1$ and that $q = p_1 \dots p_k$. Observe that $p_1 \dots p_{k-1}* \in W_k^*$ covers all prefixes with a single insertion error. Similarly, $p = p_1 \dots p_{k-s}*^s \in W_k^*$ covers all prefixes with up to s insertion errors. In general, to cover the prefixes with up to δ insertion errors with $*$-prefixes with s “don't care“ characters, we require prefixes of the form $p'*^s \in W_k^*$, where $p' \in \{p'' \in W_{k-s} \mid \text{PLD}(q, p'') \leq \delta\}$. It is easy to see that $\text{PLD}(q, p'*^s) \leq \delta$ since $\text{PLD}(q, p') \leq \delta$;
(ii) *substitution errors:* let $\{i_1, \dots, i_s\}$ be a set of position in q. Then the $*$-prefix $p = p'_1, \dots, p'_k$, where $p'_j = *$ if $j \in \{j_1, \dots, j_s\}$ and $p'_j = p_j$ otherwise, covers all prefixes with at least one substitution error on positions $\{j_1, \dots, j_s\}$. In general, to cover the prefixes with up to δ substitution errors we require $*$-prefixes in W_k^* with “don't care“ characters at any set of positions $\{j_1, \dots, j_s\}$ in q[1];
(iii) *deletion errors:* prefixes with up to δ deletion errors are covered in a similar way, however, deletion errors in a prefix of fixed length are paired with additional insertion errors with characters ahead in the string. Hence, if q contains deletion errors, then it will not be within the distance threshold from the $*$-prefixes in W_k^* anymore. For example, assume $k = 6$ and $\delta = 1$ and suppose q=`algoit` (obtained from `algori` by a deletion at positions 5) and $q \notin W_k$. It is not hard to see that q is not within distance threshold from any $*$-prefix $p \in W_k^*$. Deletion errors are instead addressed by using the $*$-prefixes from $W_k^{**} - W_k^*$ of length $k + \delta$. Assume that $m \leq s$ deletion errors have taken place on positions $\{j_1, \dots, j_m\}$. Then the $*$-prefix $p = p'_1, \dots, p'_{k+m}$, where $p'_j = *$ if $j \in \{j_1, \dots, j_m\}$ and $j \leq k$ and $p'_j = p_j$ otherwise, covers all prefixes with m deletion errors on positions $\{j_1, \dots, j_m\}$. To cover all prefixes with at most δ deletion errors, we consider $*$-prefixes with “don't care“ characters that correspond to any set of positions $\{j_1, \dots, j_s\}$, where $j_s \leq k$. It is clear that if p is defined as in (*ii*) or (*iii*), then by construction $p \in W_k^{**}$ and $\text{PLD}(q, p) \leq \delta$;
(iv) *combination of errors:* for prefixes that contain any combination of deletion and substitution or insertion and substitution errors, p is constructed analogously. However, since deletion and insertion errors have an inverse effect in q, constructing $*$-prefix p as in (*i*) and (*iii*) to address a combination of insertion and deletion errors can result in p that is not within δ from q. The reason for this is that some of the “don't care“ characters must overwrite matching characters at the end of p. For example, assume $\delta = 2$ and let q=`alxgoit` (q contains one insertion and one deletion error). By using (*i*), we obtain p=`algori*` with $\text{PLD}(q, p) = 3 > \delta$. Assume that there are m_1 deletion and m_2 insertion errors, where $m_1 + m_2 \leq \delta$. Observe that since $s \leq \lceil \delta/2 \rceil$, we can construct p by considering only $\max\{0, m_2 - m_1\}$ of the m_2 insertion errors and simply ignoring the rest. Now by using (*i*) and (*iii*) we always obtain p that is within δ from q. □

Proof of Lemma 8.3.1. Call the $s \cdot k$ random numbers picked in the beginning splitters. Let the maximum block size be B_{max}. We consider the event that B_{max} is larger than some B. Then there must be a sub-range of size B that contains strictly less that s splitters. There are $n - B + 1 \leq n$ such sub-ranges in total which means that $Pr(B_{max} > B) \leq n \cdot p$, where p is the probability that a fixed sub-range of size B contains less than s splitters. This probability is equal to

$$\sum_{i=0}^{s-1} \binom{sk}{i} (B/n)^i \, (1 - B/n)^{sk-i}$$

We will derive an upper bound on the probability $p(s)$ that exactly s splitters fall into a fixed range of size b and from there derive Equation 8.1. After plugging $B = a \cdot n/k$ into $p(s)$ and applying the inequalities $\binom{sk}{s} \leq (ek)^s$

[1] Note that in practice we must compute a minimal set of such $*$-prefixes that cover all prefixes in W (i.e., consider only the $*$-prefixes that contain at least one matching prefix that has not been covered by other $*$-prefixes).

and $1 - x < \exp(-x)$; by simple transformations we obtain

$$p(s) \leq \exp\left(\left(1 + \ln(a) - a \cdot \frac{k-1}{k}\right) \cdot s\right) \tag{A.1}$$

which can be written as $\exp(-C \cdot s)$, for $C > 0$. This inequality is satisfied for all practical values of k (e.g. $k > 1000$), provided that $a > 1$. To complete the proof we will use the inequality $p \leq s \cdot p(s)$ provided that $p(s) \geq p(s-1) \geq ... \geq p(0)$. For the binomial distribution the latter holds if s is no larger than the mode of the distribution M as $p(s)$ is maximized when $s = M$. In our case this condition is satisfied as

$$M = \lfloor sk \cdot B/n \rfloor \geq sk \cdot a/k = s \cdot a$$

which is larger than s if $a > 1$. By plugging in Equation A.1 in the latter inequality we obtain

$$p \leq \exp\left((1 + \ln(s)/s + \ln(a) - a \cdot (k-1)/k) \cdot s\right)$$

which can be also written as $\exp(-K \cdot s)$. Again, $K > 0$ for small values of $a > 1$ and all practical values of k. This concludes the proof. □

Proof of Lemma 8.4.1. Under the assumptions given in Section 8.4.1, the expected numbers of cache misses for HYB posting accumulation is equal to

$$l \cdot n \cdot \left(1 - \frac{c}{\sqrt[l]{k}}\right) \tag{A.2}$$

assuming $c \leq \sqrt[l]{k}$. Obviously, by picking $l = \lceil \log k / \log c \rceil$ we could reduce the number of cache misses to 0, however ideally one should minimize the total cost by computing

$$\operatorname*{argmin}_{l}\, l \cdot \left(\left(1 - \frac{c}{\sqrt[l]{k}}\right) \cdot T_m + \frac{c}{\sqrt[l]{k}} \cdot T_h\right) \tag{A.3}$$

where T_m and T_h respectively are the costs for a cache miss and a cache hit.

In the following we compare the total posting accumulation cost for $l = 1$ and $l = 2$. The expected numbers of cache misses for $l = 1$ and $l = 2$ are $n \cdot (1 - c/k)$ and $2n \cdot (1 - c/\sqrt{k})$, respectively. Given that the ratio between the two is equal to

$$\frac{k - c}{2\sqrt{k}(\sqrt{k} - c)} \tag{A.4}$$

the following three cases could take place when $l = 2$: first, the number of cache misses will be less for $k < 4 \cdot c^2$; second, the number of cache misses will be less by a large factor for $\sqrt{k} \sim c$; and third, the number of cache misses will be equal to zero for $\sqrt{k} \leq c$. The last scenario is realistic given that the number of blocks is typically less than 10,000 and that todays L1-caches are larger than 8 KB. Namely, 8 KB cache with 64 B cache lines, has $c = 128$ cache lines in total, where $\sqrt{k}$ is usually less than 100. By assuming that $\sqrt{k} \leq c$, the total cost for $l = 2$ is simply equal to $2 \cdot n \cdot T_h$, while the ratio between the two costs is equal to

$$\begin{aligned} &\frac{n\left(\left(1 - \frac{c}{k}\right) T_m + \frac{c}{k} \cdot T_h\right)}{2n \cdot T_h} \\ &= \frac{1}{2} \cdot \frac{T_m}{T_h} - \frac{c}{k} \cdot \frac{T_m - T_h}{2T_h} \end{aligned}$$

Say a cache hit is m times faster than a cache miss i.e. $T_m/T_h = m \geq 2$. Two-level posting accumulation is then faster by a factor of

$$\frac{m}{2} - g(k) \tag{A.5}$$

where $g(k) = \frac{c}{k} \cdot \frac{m-1}{2}$ is small and reaches its minimum for $c = \sqrt{k}$, resulting in a factor of $\frac{m}{2} - \frac{m-1}{2\sqrt{k}} \approx \frac{m}{2}$ speed-up. Indeed, the best result in practice was achieved for $l = 2$. □

Proof of Lemma 8.5.2. Observe that by using partial flushing the total number of runs will increase since the memory buffer is not always fully emptied, however the average number of disk seeks per run will decrease since only the large blocks are flushed. The memory threshold T of the small blocks on average is reached once in every

$$\lfloor T/f \rfloor + 1$$

runs, at which point $L_1 + L_2$ disk seeks are required. Every other run requires only L_1 disk seeks. The average number of disk seeks per run from $L_1 + L_2$ hence reduces to

$$\begin{aligned} \left(1 - \frac{1}{\lfloor T/f \rfloor + 1}\right) \cdot L_1 + \frac{1}{\lfloor T/f \rfloor + 1} \cdot (L_1 + L_2) \\ = L_1 + \frac{L_2}{1 + \lfloor T/f \rfloor} \end{aligned}$$

The total number of runs required will be increased by a factor of

$$\begin{aligned} \frac{\lfloor T/f \rfloor + 1}{\sum_{i=0}^{\lfloor T/f \rfloor} (1 - i \cdot f)} &= \frac{\lfloor T/f \rfloor + 1}{(\lfloor T/f \rfloor + 1)\left(1 - f \frac{\lfloor T/f \rfloor}{2}\right)} \\ &= \frac{2}{2 - f \cdot (\lfloor T/f \rfloor)} \end{aligned}$$

The total number of disk seeks is the product of the average number of seeks per run and the total number of runs. Compared to the baseline, partial flushing reduces the required number of disk seeks at least by a factor of

$$\begin{aligned} \frac{2 - T}{2} \cdot \frac{1 + \lfloor T/f \rfloor}{1 + \frac{L_1}{k} \lfloor T/f \rfloor} &\geq \frac{1}{2} \cdot \frac{fk + k}{fk + L_1} \\ &= \frac{1}{2} \cdot \frac{m + fm}{1 + fm} \end{aligned}$$

where $m = k/L_1$. □

Proof of 8.5.2. Assume that each $*$-prefix contains s “don’t care“ characters. For each combination of errors in q (deletions, insertions, substitutions) we show how to find a family of $*$-prefixes p in W_k^{**} with $\text{PLD}(q, p) \leq \delta$, such that S_q is covered. (i) insertion errors: for simplicity, assume that $\delta = 1$ and that the “intended keyword“ is $p_1...p_k$. It is not hard to see that $p_1...p_{k-1}* \in W_k^*$ covers all prefixes relatively to which a single insertion error has taken place in q (e.g. $q = p_1 x p_2 ... p_{k-1}$). In general, $p = p_1..p_{k-m+s}*^s \in W_k^*$ covers all prefixes relatively to which up to s insertion errors have taken place in q. To cover all prefixes relatively to which (up to) δ insertion errors have taken place in q, in the worst case we must consider each $*$-prefix of type $p' *^s$, where $p' \in \{p \in W_{k-s} \mid \text{PLD}(q, p) \leq \delta\}$. ($ii$) substitution errors: let $\{i_1, ..., i_s\}$ be a set of position in q. Then the $*$-prefix $p = p'_1, ..., p'_k$, where $p'_j = *$ if $j \in \{j_1, ..., j_s\}$ and $p'_j = p_j$ otherwise, covers all prefixes with at least one substitution error on position $\{j_1, ..., j_s\}$. To cover all prefixes that contain up to δ substitution errors, in the worst case we should consider all $*$-prefixes in W_k^* with “don’t care“ characters that correspond to any set of positions $\{j_1, ..., j_s\}$[2]. (iii) deletion errors: prefixes with up to δ deletion errors are covered in a similar way, however, since deletion errors additionally include characters that are ahead, a prefix q with deletion errors will not be within the distance threshold from the $*$-prefixes in W_k^* anymore. Deletion errors are instead addressed by using the $*$-prefixes from $W_k^{**} - W_k^*$ with length $k + \delta$. Assume that $m \leq s$ deletion errors have taken place in q on positions $\{j_1, ..., j_m\}$ and that the “intended keyword“ is $p_1...p_{k+m} \in W_{k+\delta}$. Then the $*$-prefix $p = p'_1, ..., p'_{k+m}$, where $p'_j = *$ if $j \in \{j_1, ..., j_m\}$ and $j \leq k$ and $p'_j = p_j$ otherwise, covers all prefixes relatively to which (at most) m deletion errors have taken place in q on positions $\{j_1, ..., j_m\}$. To cover all prefixes with at most δ deletion errors, in the worst case we have to consider all $*$-prefixes with “don’t care“ characters that correspond to any set of positions $\{j_1, ..., j_s\}$, where $j_s \leq k$. It is clear that if p is defined as in (i), (ii) and (iii), then by construction $p \in W_k^{**}$ and $\text{PLD}(q, p) \leq \delta$. For prefixes that contain any combination of deletion and substitution or insertion and substitution errors p is constructed analogously. However, since deletion and

[2] Note that in practice we need to consider only those $*$-prefixes that contain at least one matching prefix that has not been covered before

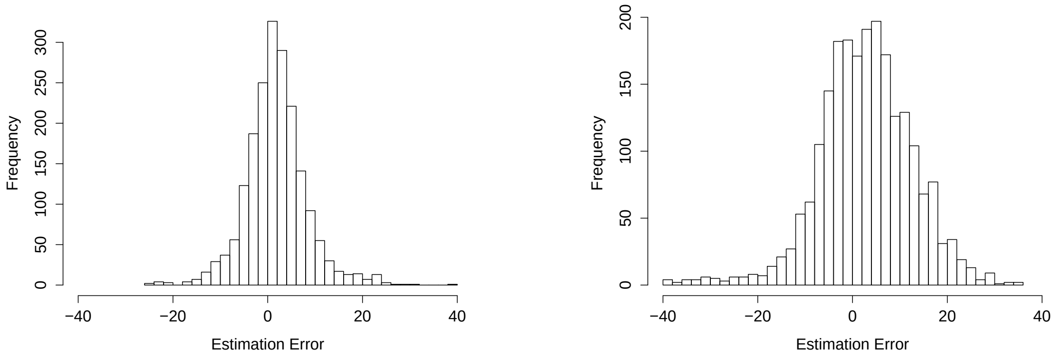


Figure A.1.: Empirical distribution of the block size estimation error ε for DBLP (left) and Wikipedia (right) when 5% of all documents are sampled.

Table A.1.: Percentage of block estimation errors within one, two and three standard deviations. According to the called 3-sigma rule for normal distribution, around 68% of the data should lie within one, about 95% within two and about 99.7% within three standard deviations.

	$\leq \sigma$	$\leq 2\sigma$	$\leq 3\sigma$
DBLP	74.5%	94.7%	98.1%
Wikipedia	73.9%	95.1%	99.2%

insertion errors have an inverse effect, constructing p as in (i) and (ii) to address a combination of insertion and deletion errors can result in p that is not within δ from q. The reason for this is that some of the "don't care" characters must overwrite matching characters at the end of p. For example, assume $\delta = 2$ and let q=`alxgoit` (q contains one insertion and one deletion error). By using (i), we obtain p=`algori*` with PLD$(q, p) = 3 > \delta$. Assume that there are m_1 deletion and m_2 insertion errors, where $m_1 + m_2 \leq \delta$. Observe that since $s \leq \lceil \delta/2 \rceil$, we can construct p by considering only $\max\{0, m_2 - m_1\}$ of the m_2 insertion errors and simply ignoring the rest. Now by using (i) and (ii) we always obtain p that is within δ from q. □

A.2. Evidence for Observation 8.5.4

To provide some theoretical evidence that the block space propagation procedure from Section 8.5.3 fails with small probability we consider the following model. Assume that the correct block size is B and that the estimated block size is $\hat{B}$. We define the estimation error of that block as

$$\varepsilon = \frac{\hat{B} - B}{B}$$

or the fraction for which the estimated block size varies from the true block size. Assume that the estimated size of each block varies around the true size with Gaussian error with mean 0, i.e. $\varepsilon \sim N(\mu, \sigma^2)$ with $\mu = 0$. To provide evidence that this model is realistic, Figure A.1 and Table A.1 show the empirical distribution of ε on two of our test collections. Furthermore, assume that $1 - r$ is a fraction of the full collection that is large enough to provide a reliable estimation for the true block sizes (e.g. 0.8 or 80%). Since the space propagation goes in discrete steps, we assume that there are enough of them for the propagation to "converge". We will compute an upper bound on the probability that at least one space propagation failures takes place.

Consider the event that a single block causes a space propagation failure on its own. This event will occur if the block size is underestimated to the degree that the maximum borrowable space (from both of its neighbors) is insufficient to fit its data:

$$B \cdot \varepsilon + 2 \cdot r < 0$$

where ε is the estimation error of that block. Note that by adding another block, the probability of the latter

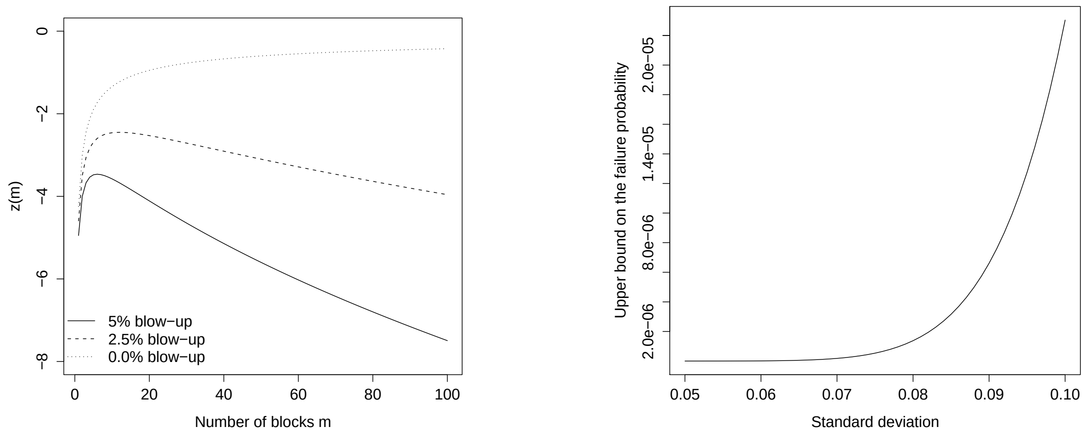


Figure A.2.: Left: value of the $z(m)$ argument of the space propagation failure probability with $r = 70\%$ and 3 block space blow-up factors. (note that already $p(-4) \sim 7.7 \cdot 10^{-9}$). Right: Upper bound on the space propagation failure probability when the standard deviation (σ) grows from 5% to 10% (the space blow-up is set to be equal to the standard deviation).

isolated event does not change. However, even if both blocks do not cause a propagation failure on their own, they can still cause a joint propagation failure when their space demand is combined. This event will occur if

$$B \cdot \varepsilon_1 + B \cdot \varepsilon_2 + 2 \cdot r < 0$$

where ε_1 and ε_2 are the estimation errors of the first and the second block, respectively (note that the blocks can borrow space from each other). Let $F_{i,j}$ for $i \le j$ be the event that the group of blocks that starts at position i and ends at position j, causes a (combined) propagation failure. The probability of at least one propagation failure (or just a failure probability) is then equal to

$$Pr(\cup_{1 \le i \le j \le k} F_{i,j})$$

Let $p_{j-i} = Pr(F_{i,j})$. Since $Pr(F_{i_1,j_1}) = Pr(F_{i_2,j_2})$ for $j_2 - i_2 = j_1 - i_1$, by the union bound we obtain

$$Pr\left(\cup_{1 \le i \le j \le k} F_{i,j}\right) \le k \cdot p_1 + (k-1) \cdot p_2 + ... + 1 \cdot p_k \tag{A.6}$$

The question now is whether the joint failure probability for a group of m blocks (p_m) increases when m ($\le k$) grows. The joint propagation failure probability for a group of m blocks is equal to

$$p_m = Pr\left(\sum_{i=1}^{m} \varepsilon_i + 2 \cdot r < 0\right) \tag{A.7}$$

According to the assumptions, $\sum_{i=1}^{m} \varepsilon_i$ has Gaussian distribution with mean $m \cdot \mu$ and variance $m \cdot \sigma^2$, which means that p_m can be written as

$$p_m(z) = \frac{1}{2\left(1 + \operatorname{erf}\left(z\left(m\right)\right)\right)}$$

where erf() is the Gaussian error function and

$$z(m) = \frac{-2r - m\mu}{\sigma\sqrt{2m}} \tag{A.8}$$

Note that $p_m(z)$ is a strictly increasing function of $z(m)$. Obviously if $\mu = 0$, then $z(m)$ strictly increases with m, resulting in large values of $p_m(z)$ (e.g. 0.5 for $m = 2000$). However, if $\mu > 0$ (a small blow-up in the block sizes), then $z(m)$ reaches a maximum for $m = 2 \cdot r/\mu$ and then starts to decrease, resulting in extremely small values of $p_m(z)$ for large m (e.g. $m = 2000$). Figure A.2 (left) plots the $z(m)$ value against the number of blocks

m with three different space blow-up factors.

For reasonable values of the standard deviation of the block size estimation error σ, the upper bound on the failure probability given in Equation A.6 remains small, e.g. for any given number of blocks, assuming $r = 30\%$, the failure probability ranges from $2.5 \cdot 10^{-9}$ to $2.3 \cdot 10^{-5}$ when $5\% \leq \sigma \leq 10\%$ as Figure A.2 (right) shows.

Bibliography

Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana deOliveira. Characterizing reference locality in the www. Technical report, Boston University, 1996.

Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR '06)*, pages 372–379, New York, NY, USA, 2006. ACM.

Ricardo A. Baeza-Yates and Gaston H. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware (SPIRE '99)*, pages 16–, Washington, DC, USA, 1999. IEEE Computer Society.

Ricardo Baeza-yates and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *In Proceeding of the International Symposium on String Processing and Information Retrieval (SPIRE '98)*, pages 14–22, Heidelberg, Germany, 1998. Springer-Verlag.

Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. Query recommendation using query logs in search engines. In *International Workshop on Clustering Information over the Web (ClustWeb '04)*, volume 3268 of *Lecture Notes in Computer Science*, pages 588–596, Creete, Greece, 2004. Springer.

Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. *Lecture Notes in Computer Science*, 3109:400–408, 2004.

Ranieri Baraglia, Carlos Castillo, Debora Donato, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Aging effects on query flow graphs for query suggestion. In *Proceeding of the 18th Conference on Information and Knowledge Management (CIKM '09)*, pages 1947–1950, New York, NY, USA, 2009. ACM.

Hannah Bast and Marjan Celikik. Efficient two-sided error-tolerant search. In *Proceedings of the 2nd International Workshop on Keyword Search on Structured Data (KEYS '10)*, page 3, Indianapolis, Indiana, USA, 2010. ACM.

Hannah Bast and Marjan Celikik. Fast construction of the hyb index. *ACM Transaction of Information Systems*, 29, July 2011.

Holger Bast and Ingmar Weber. Type less, find more: fast autocompletion search with a succinct index. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR '06)*, pages 364–371, New York, NY, USA, 2006. ACM.

Holger Bast and Ingmar Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *Third Conference on Innovative Data Systems Research (CIDR'07)*, pages 88–95, Asilomar, CA, USA, 2007. VLDB Endowment.

Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Databases (VLDB '06)*, pages 475–486, New York, USA, 2006. VLDB Endowment.

Holger Bast, Alexandru Chitea, Fabian Suchanek, and Ingmar Weber. Ester: efficient search on text, entities, and relations. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 671–678, New York, NY, USA, 2007. ACM.

Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*, pages 131–140, New York, NY, USA, 2007. ACM.

Djamal Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM '09)*, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag.

Sumit Bhatia, Debapriyo Majumdar, and Prasenjit Mitra. Query suggestions in the absence of query logs. In *Proceedings of the 34th International Conference on Research and Development in Information (SIGIR '11)*, pages 795–804, New York, NY, USA, 2011. ACM.

Paolo Boldi, Francesco Bonchi, Carlos Castillo, Debora Donato, and Sebastiano Vigna. Query suggestions using query-flow graphs. In *Proceedings of the Workshop on Web Search Click Data (WSCD '09)*, pages 56–63, New York, NY, USA, 2009. ACM.

Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics*, 16:1.1:1.1–1.1:1.91, May 2011.

Paul Bratley and Yaacov Choueka. Processing truncated terms in document retrieval systems. *Information Processing and Management*, 18(5):257–266, 1982.

Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99)*, pages 126–134, New York, NY , USA, 1999. IEEE Computer Society Press.

Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics (ACL '00)*, pages 286–293, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

Chris Buckley. Why current IR engines fail. In *Proceedings of the 27th Annual International Conference on Research and Development in Information Retrieval (SIGIR '04)*, pages 584–585, Hingham, MA, USA, 2004. ACM.

Stefan Buettcher. The Wumpus search engine, 2007. http://www.wumpus-search.org/index.html.

Stefan Buttcher and Charles L. A. Clarke. Memory management strategies for single-pass index construction in text retrieval systems. Technical report, School of Computer Science, University of Waterloo, Canada, 2005.

Stefan Büttcher and Charles L. Clarke. Hybrid index maintenance for contiguous inverted lists. *Information Retrieval*, 11(3):175–207, 2008.

Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. Context-aware query suggestion by mining click-through and session data. In *Proceeding of the 14th International Conference on Knowledge Discovery and Data Mining (KDD '08)*, pages 875–883, New York, NY, USA, 2008. ACM.

Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Caching query-biased snippets for efficient retrieval. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT '11)*, pages 93–104, New York, NY, USA, 2011. ACM.

Marjan Celikik and Hannah Bast. Fast single-pass construction of a half-inverted index. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE '09)*, pages 194–205, Berlin, Heidelberg, 2009. Springer-Verlag.

Marjan Celikik and Holger Bast. Fast error-tolerant search on very large texts. In *Symposium of Applied Computing (SAC '09)*, pages 1724–1731, New York, NY, USA, 2009. ACM.

Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In *Proceedings of the 35th International Conference on Management of Data (SIGMOD '09)*, pages 707–718, New York, NY, USA, 2009. ACM.

Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computational Surveys*, 33(3):273–321, 2001.

Charles L. A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Transaction of Information Systems*, 18(1):44–78, January 2000.

C. L. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

Charles L. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38:43–56, 1995.

Archie L Cobbs. Fast approximate matching using suffix trees. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, page 4154, 1995.

William G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley & Sons, Inc., New York, USA, 1977.

Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (STOC '04)*, pages 91–100, New York, NY, USA, 2004. ACM.

Silviu Cucerzan and Ryen W. White. Query suggestion based on user landing pages. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 875–876, New York, NY, USA, 2007. ACM.

D. et al Cutting. Lucene, 2004. http://lucene.apache.org.

Raymond J. D'Amore and Clinton P. Mah. One-time complete indexing of text: theory and practice. In *Proceedings of the 8th Annual International Conference on Research and Development in Information Retrieval (SIGIR '85)*, pages 155–164, New York, NY, USA, 1985. ACM.

Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM symposium on Discrete Algorithms (SODA '00)*, pages 743–752, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

M. W. Du and S. C. Chang. An approach to designing very fast approximate string matching algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):620–633, August 1994.

Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth Symposium on Principles of Database Systems (PODS '01)*, pages 102–113, New York, NY, USA, 2001. ACM.

Paolo Ferragina and Rossano Venturini. Compressed permuterm index. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 535–542, New York, NY, USA, 2007. ACM.

Karina Figueroa, Edgar Chávez, Gonzalo Navarro, and Rodrigo Paredes. On the least cost for proximity searching in metric spaces. In *Proceedings of the 5th Workshop on Efficient and Experimental Algorithms (WEA '06)*, Lecture Notes in Computer Science, pages 279–290, Cala Galdana, Menorca, Spain, 2006. Springer.

Wei Gao, Cheng Niu, Jian-Yun Nie, Ming Zhou, Jian Hu, Kam-Fai Wong, and Hsiao-Wuen Hon. Cross-lingual query suggestion using query logs of different languages. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 463–470, New York, NY, USA, 2007. ACM.

Jade Goldstein, Mark Kantrowitz, Vibhu Mittal, and Jaime Carbonell. Summarizing text documents: sentence selection and evaluation metrics. In *Proceedings of the 22nd Annual International Conference on Research and Development in Information Retrieval (SIGIR '99)*, pages 121–128, New York, NY, USA, 1999. ACM.

Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, Boston, MA, USA, January 2003.

Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 491–500, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.

Harold Stanley Heaps. *Information Retrieval – Computational and Theoretical Aspects*. Academic Press, 1978.

Steffen Heinz and Justin Zobel. Performance of data structures for small sets of strings. *Australian Computer Science Communications*, 24:87–94, 2002.

Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54:713–729, 2003.

Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54:713–729, 2003.

Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

E. B. James and D. P. Partridge. Adaptive correction of program statements. *Communications of the ACM*, 16(1):27–37, January 1973.

Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*, pages 371–380, New York, NY, USA, 2009. ACM.

Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In Petteri Jokinen and Esko Ukkonen, editors, *In Proceedings of the 2nd Annual Symposium on Mathematical Foundations of Computer Science*, volume 520, pages 240–248, 1991.

Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 351–360, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

Youngho Kim, Jangwon Seo, and W. Bruce Croft. Automatic boolean query suggestion for professional search. In *Proceedings of the 34th International Conference on Research and Development in Information Retrieval (SIGIR '11)*, pages 825–834, New York, NY, USA, 2011. ACM.

Youngjoong Ko, Hongkuk An, and Jungyun Seo. An effective snippet generation method using the pseudo relevance feedback technique. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 711–712, New York, NY, USA, 2007. ACM.

Juha Kken and Joong Chae Na. Faster filters for approximate string matching. In *ALENEX*. SIAM, 2007.

Karen Kukich. Technique for automatically correcting words in text. *ACM Computing Surveys*, 24:377–439, 1992.

Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of the 6th International Conference on Web Information Systems Engineering (WISE '05)*, pages 470–477, Berlin, Heidelberg, 2005. Springer-Verlag.

V I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

Mu Li, Yang Zhang, Muhua Zhu, and Ming Zhou. Exploring distributional similarity based models for query spelling correction. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics (ACL '06)*, pages 1025–1032, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the 24th International Conference on Data Engineering (ICDE '08)*, pages 257–266, Washington, DC, USA, 2008. IEEE Computer Society.

Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. Efficient fuzzy full-text type-ahead search. *The VLDB Journal*, 20(4):617–640, 2011.

Wentian Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38:1842–1845, 1992.

David E. Losada, Leif Azzopardi, and Mark Baillie. Revisiting the relationship between document length and relevance. In *Proceedings of the 17th Conference on Information and Knowledge Management (CIKM '08)*, pages 419–428, New York, NY, USA, 2008. ACM.

Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC '93)*, pages 286–293, New York, NY, USA, 1993. ACM.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

Qiaozhu Mei, Dengyong Zhou, and Kenneth Church. Query suggestion using hitting time. In *Proceeding of the 17th Conference on Information and Knowledge Management (CIKM '08)*, pages 469–478, New York, NY, USA, 2008. ACM.

Christian Middleton and Ricardo Baeza-Yates. A comparison of open source search engines, 2007.

Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30:451–477, December 2004.

Alistair Moffat and Timothy A. H. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science and Technology*, 46:537–550, 1995.

Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transaction of Information Systems*, 14(4):349–379, 1996.

Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions of Information Systems*, 14:349–379, 1996.

Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR '06)*, pages 348–355, New York, NY, USA, 2006. ACM.

M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Communications of the ACM*, 25(12):935–938, December 1982.

Robert Muth and Udi Manber. Approximate multiple strings search. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching (CPM '96)*, volume 1075 of *Lecture Notes in Computer Science*, pages 75–86, Laguna Beach, California, USA, 1996. Springer.

E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, V12(4):345–374, October 1994.

Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46:1–13, 1999.

Arnab Nandi and H. V. Jagadish. Effective phrase prediction. In *Proceedings of the 33rd International Conference on Very large Databases (VLDB '07)*, pages 219–230. VLDB Endowment, 2007.

G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998.

Gonzalo Navarro and Ricardo Baeza-yates. A hybrid indexing method for approximate string matching. *Area*, 0(0):1–35, 2000.

Gonzalo Navarro and Leena Salmela. Indexing variable length substrings for exact and approximate matching. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE '09)*, pages 214–221, Berlin, Heidelberg, 2009. Springer-Verlag.

Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.

Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, and Jorma Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (COM '00)*, pages 350–363, London, UK, UK, 2000. Springer-Verlag.

Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

Florentina I. Popovici, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau. Robust, portable I/O scheduling with the disk mimic. In *USENIX Annual Technical Conference*, pages 297–310, San Antonio, TX, USA, 2003. IEEE.

Eric Sven Ristad and Peter N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:522–532, 1998.

Willie Rogers, Gerald Candela, and Donna Harman. Space and time improvements for indexing in information retrieval. In *Proceedings of 4th Annual Symposium on Document Analysis and Information Retrieval (SDAIR '95)*, Las Vegas, USA, 1995. University of Nevada.

Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Approximate string matching with lempel-ziv compressed indexes. In *Proceedings of the 14th International Conference on String Processing and Information Retrieval (SPIRE '07)*, pages 264–275, Berlin, Heidelberg, 2007. Springer-Verlag.

Lu. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

David Sankoff. Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A.*, 69:4–6, 1972.

Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International Conference on Research and Development in Information Retrieval (SIGIR '02)*, pages 222–229, New York, NY, USA, 2002. ACM.

Klaus U Schulz and Stoyan Mihov. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85, 2002.

Peter H. Sellers. On the Theory and Computation of Evolutionary Distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

Fei Shi and Cread Mefford. A new indexing method for approximate search in text databases. In *Proceedings of the The Fifth International Conference on Computer and Information Technology (CIT '05)*, pages 70–76, Washington, DC, USA, 2005. IEEE Computer Society.

Yang Song and Li-wei He. Optimal rare query suggestion with implicit user feedback. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, pages 901–910, New York, NY, USA, 2010. ACM.

Erkki Sutinen and Jorma Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of the Third Annual European Symposium on Algorithms (ESA '95)*, pages 327–340, London, UK, 1995. Springer-Verlag.

Erkki Sutinen and Jorma Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM '96)*, pages 50–63, London, UK, 1996. Springer-Verlag.

Erkki Sutinen and Jorma Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM '96)*, pages 50–63, London, UK, UK, 1996. Springer-Verlag.

The National Archives. The soundex indexing system, May 2007.

Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth International conference on Very Large Databases (VLDB '04)*, pages 648–659. VLDB Endowment, 2004.

Anastasios Tombros and Mark Sanderson. Advantages of query biased summaries in information retrieval. In *Proceedings of the 21st Annual International Conference on Research and Development in Information Retrieval (SIGIR '98)*, pages 2–10, New York, NY, USA, 1998. ACM.

Yohannes Tsegay, Simon J. Puglisi, Andrew Turpin, and Justin Zobel. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval (ECIR '09)*, pages 509–520, Berlin, Heidelberg, 2009. Springer-Verlag.

Andrew Turpin, Yohannes Tsegay, David Hawking, and Hugh E. Williams. Fast generation of result snippets in web search. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR '07)*, pages 127–134, New York, NY, USA, 2007. ACM.

Essko Ukkonen. Algorithms for approximate string matching. *Information and Control 64*, 158, 1983.

Esko Ukkonen. Approximate string-matching over suffix trees. *Lecture Notes in Computer Science*, 684:228242, 1993.

Ramakrishna Varadarajan and Vagelis Hristidis. A system for query-specific document summarization. In *Proceedings of the 15th Conference on Information and Knowledge Management (CIKM '06)*, pages 622–631, New York, NY, USA, 2006. ACM.

T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.

Dingding Wang, Tao Li, Shenghuo Zhu, and Chris Ding. Multi-document summarization via sentence-level semantic analysis and symmetric matrix factorization. In *Proceedings of the 31st Annual International Conference on Research and Development in Information Retrieval (SIGIR '08)*, pages 307–314, New York, NY, USA, 2008. ACM.

Jiannan Wang, Guoliang Li, Jianhua Feng, and Chen Li. Automatic url completion and prediction using fuzzy type-ahead search. In *Proceedings of the 32nd International Conference on Research and Development in Information Retrieval (SIGIR '09)*, pages 634–635, New York, NY, USA, 2009. ACM.

Ryen W. White, Ian Ruthven, and Joemon M. Jose. Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In *Proceedings of the 25th Annual International Conference on Research and Development in Information Retrieval (SIGIR '02)*, pages 57–64, New York, NY, USA, 2002. ACM.

P. Willett and R.C. Angell. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management*, 19(4):255–261, 1983.

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Diego, CA, USA, 1999.

Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of ACM*, 35(10):83–91, October 1992.

Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of The VLDB Endowment*, 1:933–944, 2008.

Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceeding of the 17th International Conference on World Wide Web (WWW '08)*, pages 131–140, New York, NY, USA, 2008. ACM.

Justin Zobel and Philip Dart. Finding approximate matches in large lexicons. *Software Practice and Experience*, 25(3):331–345, March 1995.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computer Surveys*, 38:6, 2006.

Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80:271–277, 2001.