Efficient Fuzzy Search in Large Text Collections

HANNAH BAST and MARJAN CELIKIK, Albert Ludwigs University

We consider the problem of fuzzy full-text search in large text collections, that is, full-text search which is robust against errors both on the side of the query as well as on the side of the documents. Standard inverted-index techniques work extremely well for ordinary full-text search but fail to achieve interactive query times (below 100 milliseconds) for fuzzy full-text search even on moderately-sized text collections (above 10 GBs of text). We present new pre-processing techniques that achieve interactive query times on large text collections (100 GB of text, served by a single machine). We consider two similarity measures, one where the query terms match similar terms in the collection (e.g., `algorithm` matches `algoritm` or vice versa) and one where the query terms match terms with a similar prefix in the collection (e.g., `alori` matches `algorithm`). The latter is important when we want to display results instantly after each keystroke (search as you type). All algorithms have been fully integrated into the CompleteSearch engine.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Query processing; H.2.4 [**Systems**]: Textual databases; H.2.8 [**Database Applications**]: Scientific databases; H.3.3 [**Information Search and Retrieval**]; H.3.4 [**Performance evaluation**]: Efficiency and effectiveness

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Approximate Dictionary Search, Approximate Text Search, Error Tolerant Autocomplete, Fuzzy Search, HYB Index, Inverted Index

ACM Reference Format:
Hannah Bast and Marjan Celikik. 2013. Efficient Fuzzy Search in Large Text Collections. ACM Trans. Inf. Syst. 9, 4, Article 39 (March 2010), 60 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

It is desirable that a search engine be robust against mistakes in the query. For example, when a user types `fuzy search alorithm`, the search engine should also return documents containing the words `fuzzy search algorithm`. Web search engines like Google provide this features by asking back "Did you mean: fuzzy search algorithm?" 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.

In many applications, it is equally desirable that the search engine is robust against mistakes on the side of the searched documents. For example, when searching for `algorithm` in a collection of computer science articles, we would also like to find those articles where that word has been misspelled, for example, as `alogorithm` or `alogritm` or `aigorithm`. This happens surprisingly often, either due to typos caused by the author of the paper, or because of OCR errors when scanning the articles from paper. Web search engines like Google do not provide this feature. The main reason is that

Author's addresses: H. Bast and M. Celikik, Department for Computer Science, Albert Ludwigs University, Freiburg, Germany.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

in web search, for a correct(ed) query, we usually have more than enough hits and the primary problem is to find the best such hits (that is, achieve good precision, especially among the top-ranked hits) and not so much to find even more hits (that is, increase the recall). However, in vertical search, like in the just mentioned literature search, recall is as important as precision and sometimes even more important. For example, if there are only a dozen of papers about fuzzy search algorithms in our database, we certainly do not want to miss the one with the misspelling `alogorithm` in the title.

The straightforward solution to the latter problem (errors on the side of the text collection, later referred to as the *intersection of union lists problem*) goes in two steps.

(1) For each query word, compute all similar words from the dictionary of the collection, that is, the set of words that occur at least once in the text collection;
(2) Replace each query word by a disjunction of all these similar words, for example, replace the query `algorithm` by `alogorithm OR alogirtm OR aigorithm OR ...`

Even with an efficient solution for (1), step (2) is inefficient when done in the straightforward way because a large number of inverted lists (hundreds or even thousands) have to be read and processed. For example, with the open source engine Lucene, the fuzzy search query `probablistic databases` takes roughly 20 seconds on the September 2010 dump of the English Wikipedia (23GB of XML, which is mostly the text of the articles).

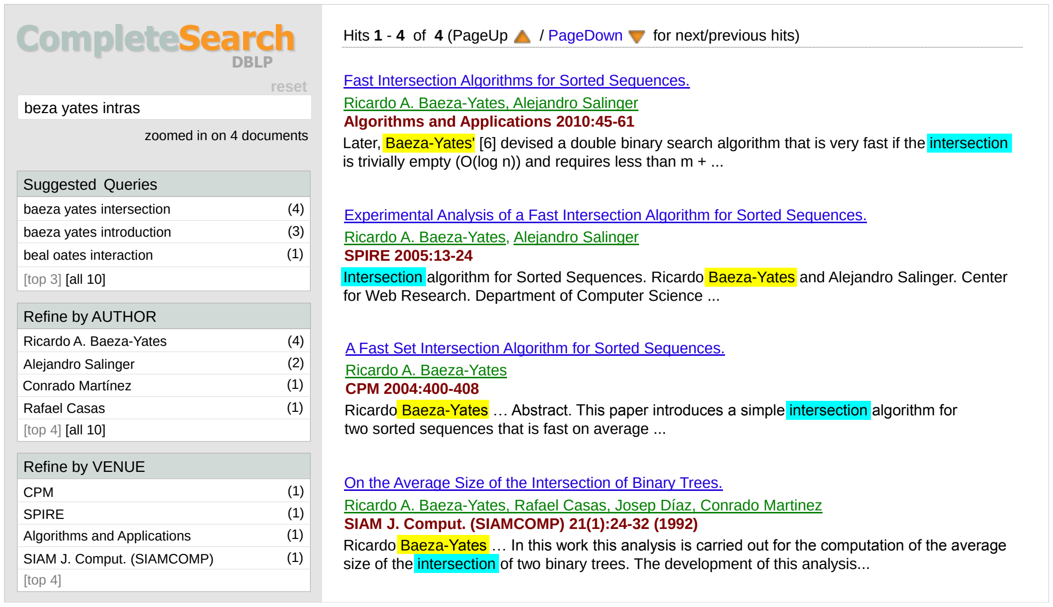
An additional complication arises when we want to support an instant display of search results along with every keystroke, as we do in this paper. Then for each query word (or at least for the query word that is currently being typed) we should find not only all words that are similar to this query word but also all words that have a prefix that is similar to that query word. For example, for the query `probabi` we should also find documents containing the word `probalistic`, because its prefix `probali` is similar to the prefix `probabi` of the query word (but not to the whole word `probabilistic`, which is a Levenshtein distance of 5 away [Levenshtein 1966]).

Google supports this feature as far as errors on the side of the query are concerned. For example, when typing `probali`, Google will suggest `probability`. However, this feature only works when the typed query is similar (in the prefix sense explained above) to a query from a (large but necessarily limited) pre-compiled list of queries. For example, when typing `probalistic datab` Google will suggest `probabilistic database`, because that is one of the pre-compiled queries. But when typing `probalistic dissi`, the suggestion is not `probabilistic dissimilarity` (although a number of research articles indexed by Google contain this phrase in their title), because this query is not in the pre-compiled list. This snapping to popular / frequent queries from a pre-compiled list makes a lot of sense for web search, but is unsatisfactory for many vertical search applications where "expert queries" with small hit sets are the rule rather than the exception.

1.1. Our contribution

In this article we present new algorithms and index data structures for a 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 the word-based and the prefix-based similarity measure mentioned above, (4) does not rely on a pre-compiled list of queries, and (5) has interactive query times (below 100 msecs) even on large text collections (up to 100 GB in size) on a single state-of-the-art machine.

We provide experiments on three datasets of distinctly different kinds and sizes: the full text of about 30,000 computer science articles from DBLP (about 1GB of text, with lots of OCR errors), the English Wikipedia (3.7 million articles, 21GB of text of relatively high quality), and the TREC GOV2 collection (25 million articles, 426GB in



beza yates intras

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...

Fig. 1: A current version of CompleteSearch doing fuzzy search with one of our new algorithms on DBLP. The query being typed is *beza yates intras* (the intended query is *baeza yates intersection*). It should be noted that Google currently does not provide any meaningful suggestion for this query.

total, with around 150GB of text, web-style content). We compare our algorithms to the straightforward approach mentioned above (disjunction of all similar words), to the previously best algorithm by Li et al. [2008], to fuzzy search with a current version of the open-source engine Lucene[1], and to a state-of-the-art exact search.

We have fully integrated our algorithms into the CompleteSearch engine [Bast and Weber 2007], which offers search results and query suggestions as you type. In previous versions of CompleteSearch, suggestions where only for the query word currently being typed. We have now extended this to suggestions for whole queries; this is explained in Section 6.7. All our algorithms can be run in two modes: (A) present the union of all hits for all fuzzy matches of the query currently being typed; or (B) while typing, present the hits only for the top-ranked query suggestion, and show the hits for the other suggestions only when they are clicked on. Figure 1 shows a screenshot of a current version of CompleteSearch with one of our new algorithms in action; the mode used here is (B).

2. PROBLEM DEFINITION AND RESULT OVERVIEW

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 paper. In Section 2.2, we define our family of problems formally. Finally, Section 2.3 provides an overview of the results for each problem.

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

Table I: 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

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 $w_1 \preceq w_2$. $\mathrm{LD}(q, w)$ 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 (*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. [2000].

Definition 2.2 (*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 $\mathrm{PLD}(p, w) \leq \delta$, we will informally say that w is a *fuzzy completion* of p. Figure I 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 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 2.3 (*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 Section 4. Efficient fuzzy autocompletion matching is dealt with in Section 5.

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 Section 6.7 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 2.4 (*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 Section 6.7. We are now ready to define our central problem.

Definition 2.5 (*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 $D' = \{d \mid d \in D, 1 \leq \forall i \leq l, \exists q_i^{'} \in Q_i$ such that $q_i^{'} \in d\}$ is the set of matching documents ranked by their relevance to $\mathcal{Q}$, 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 $D' = \{d \mid d \in D, \exists q_i^{'} \in Q_i$ such that $q_i^{'} \in d\}$ ranked by their relevance to $\mathcal{Q}$.

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 Section 6. We would like to emphasize that in our definition of fuzzy prefix search, any of the query words can be specified only partially.[2] We believe that this approach can save additional typing effort to the user as shown in the following example.

Example 2.6. 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 2.5, 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. Result Overview

In this work, we present new algorithms for all four problems defined above: fuzzy word matching, fuzzy prefix matching, fuzzy word search, fuzzy prefix search.

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 an algorithm with practical index that 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 δ, improve the previously best algorithms for up to one order of magnitude. These algorithms are explained in Sections 4.1 and 4.2, respectively, and experimental results are provided in Section 7.2.

[2]In the existing literature, prefix or look-ahead search is performed only on the last query word.

For the fuzzy prefix matching problem, we extend the aforementioned algorithms and achieve a significant improvement on short prefixes and an improvement of more than one order of magnitude on long prefixes for non-incremental search over the previous best algorithm from Ji et al. [2009]. These algorithms are described in Sections 5.1 and 5.2, respectively. We also provide a simple incremental algorithm that can be sufficiently fast in practice. This algorithm is explained in Section 5.3. Our experimental results for these algorithms are provided in Section 7.3.

For both the fuzzy word and fuzzy prefix search problem, we propose two novel data structures, called fuzzy-word and fuzzy-prefix index (explained in Sections 6.2 and 6.3, respectively) and a new query processing algorithm (explained in Section 6.4).

Our algorithms improve over our baseline algorithm by a factor of up to 7 when the index is in memory and by a factor of up to 4 when the index resides on disk. The experimental results are provided in Sections 7.4 and 7.5, respectively.

We want to stress that we took care to implement our baseline algorithm efficiently and that it was not easy to beat on larger collections; see Section 6. In fact, as we show in our experiments, the previous best algorithms for fuzzy prefix search and fuzzy word search by Ji et al. [2009], *lose* against our baseline on larger test collections. The algorithm by Ji et al. [2009] was primarily designed for exact prefix search and for fuzzy search has a query time complexity linear in the number of similar words / prefixes. For our baseline this dependency is logarithmic and for our new algorithm close to constant. See Sections 3.3 and 7.4 for more information.

Finally we propose an algorithm to compute a ranked list of query suggestions. This algorithm makes use of the result lists from the fuzzy word or prefix search algorithm. It takes only a small fraction of the total query processing time.

3. RELATED WORK

3.1. Fuzzy Word Matching

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. 2008; Xiao et al. 2008]
— 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. 2008; Xiao et al. 2008; 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.[3] 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.

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 done in Boytsov [2011]. A deletion-neighborhood based algorithm is covered in greater detail in Section 4.1.

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 2009], 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. Fuzzy Prefix Matching

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

[3]`http://flamingo.ics.uci.edu/`

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 $p_1 \ldots p_n$ for $n > 0$, we first have to compute the active nodes for the prefix $p_1 \ldots p_{n-1}$. 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]).

3.3. 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 (explained in more details later on) 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 7.4).

3.4. Query Suggestions

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[4] offered exact completions from a pre-compiled list of popular queries. Over the years, the service was significantly extended. 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.[5]

The query `beza yates intre` from our example screenshot from Figure 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.

[4]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.
[5]We say this, in particular, from our experience with running CompleteSearch DBLP for 5 years now, which gets around 5 million hits every month.

4. EFFICIENT FUZZY WORD MATCHING

This section 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 (`DeleteMatch` and `PermuteScan`) with two different trade-offs that are suited for a dynamic distance threshold (Section 2). 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.

4.1. 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 4.1.1). 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 4.1.2). 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 4.1.3 and 4.1.4).

4.1.1. 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 4.1. 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 4.2. The n-deletion neighborhood $U_d(w, n)$ of a word w consists of all k-subsequences of w, for $k = 0 \ldots n$

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

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 4.3. *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. □

Table II: Average number of distance computations per match for different query word lengths and different thresholds δ by using the basic *DeleteMatch* algorithm in Section 4.1.2 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

4.1.2. The Mor-Fraenkel Method. Given a dictionary W, a threshold δ and a query word q, Lemma 4.3 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

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

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.

4.1.3. 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].[6] 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.

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 II 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. The following lemma will allow us to index the k-truncated dictionary W^k instead of the full dictionary W.

LEMMA 4.4. *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 4.3. 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. The size of the subsequence dictionary is now smaller for at least a factor of $|W|/|W^k| \cdot (m/k)^\delta$. To reduce its size further, we will make use of the fact that our dictionary consists mainly of k-prefixes. The following lemma will allow us to index only subsequences of length $k - \delta$ instead of the full deletion neighborhoods.

LEMMA 4.5. *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 4.3 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

[6]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 $\delta = 2$.

must be a corresponding delete position in p_2 as otherwise the length of the resulting subsequences will not be equal. $\square$

It is not hard to see the efficiency of the algorithm is not affected. On the contrary, many subsequences are not generated and scanning their inverted lists is omitted. If δ is dynamic and depends on $|q|$, we can index only the subsequences of length $k - \delta(|w|)$ of each k-prefix.

We refer to the resulting set of subsequences as the *truncated deletion neighborhood* of W. It should be noted that if W is "prefix dense", then we could simply index the truncated deletion neighborhood on the reversed version of the dictionary. The size of the subsequence dictionary of the truncated deletion neighborhood in practice is very small, typically a fraction of W. This allows us storing it uncompressed in a conventional data structure for fast access such as hash table or a trie. 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	**gaffe**o	**gaffe**r	**gaffe**t	**gaffe**y	**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 III). 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.[7] Instead of using a fixed k, a better strategy in practice is choosing a larger k on longer words and a smaller k on shorter words. We did not experiment extensively with this heuristic.

4.1.4. 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 1996] 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

[7]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 7.2.

Table III: 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 IV: 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

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 V 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 also known as *unigram frequency distance*. 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 4.6. *Assume* $\mathrm{WLD}(q, w) \leq \delta$ *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* $l + 1$ *must share at least* $\max\{|w|, |q|\} - l - \delta$ *characters in common.*

PROOF. Assume that the k-prefixes of w and q are "aligned" with respect to WLD (see the proof of Lemma 4.4). Assume also that $l < k$ and that the k-prefixes of q and w share $k - l$ characters in common with position less than $l + 1$. However, this implies that q and w have equal characters on positions $l + 1 \ldots k$ (although they are counted twice) and hence the suffixes starting at position $l+1$ will still have $\max\{|w|, |q|\} - l - \delta$ characters in common. The proof when w and q have k-prefixes that are not "aligned" is similar to that of Lemma 4.4. Namely, a matching substring between q and w might have a starting position less than k in one of the words and larger than k in the other. However, both starting positions must be larger than l. □

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

Table V: 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

LEMMA 4.7. *If* $\mathrm{WLD}(q, w) \leq \delta$*, 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 $n > \delta$ characters that are not in q then any $w[j], j \geq i$ will contain the same n characters 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 characters between q and w computed so far and let F_q be the frequency vector of q and let l be the length of the current matching subsequence between q and a word w. We combine the above observations in a simple filter as follows:

(1) Initially compute F_q and set $count$ to l;
(2) For $i = l + 1 \ldots |w|$, 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 $\mathrm{WLD}(w, q) > \delta$;
(4) If the final value of $count < \max\{|q|, |w|\} - \delta$, conclude that $\mathrm{WLD}(w, q) > \delta$;
(5) Restore F_q to its previous state by increasing $F_q[w[i]]$ by 1 for $i = l + 1 \ldots j$, where j is the last value of i in the loop in step 2.

4.2. 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 4.2.1). 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 4.2.2). We then discuss which problem instances are hard for this algorithm and propose further optimizations to improve its running time (Sections 4.2.3, 4.2.4 and 4.2.5).

4.2.1. The Longest Common Substring as a Signature. The main observation in this section is that if $\mathrm{WLD}(w_1, w_2) \leq \delta$, 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 4.8. *If* $\mathrm{WLD}(q, w) \leq \delta$*, then* q *and* w *must share a substring of length at least* $\lceil \max\{|q|, |w|\}/\delta \rceil - 1$*.*

Table VI: Differences between the required common substring lengths between a query q and a word w for varying lengths dictated by Property 4.9 (longest common substring) and Property 4.10 (pattern partitioning). 0 means that both properties require equal common substring lengths.

$\|q\|$	**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

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 $\lceil n/\delta \rceil - 1$. $\square$

The following property is a direct consequence of Lemma 4.8

PROPERTY 4.9. *If* $\mathrm{WLD}(q, w) \leq \delta$*, 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 4.10. *Let* $\mathrm{WLD}(q, w) \leq \delta$*, 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.[8] One obvious difference to Lemma 4.8 is that for $\delta = 1$, Lemma 4.8 requires a common substring of length at least $|q| - 1$, whereas Property 4.10 requires a common substring of length at least $\frac{|q|}{2}$. This makes Lemma 4.9 more effective for $\delta = 1$. For many instances Lemma 4.8 requires a longer common substring because q and w can be considered as cyclic strings. Table VI compares the differences between the substring lengths for different lengths of q. For example, if $|q| = 7$ and $\delta = 2$, then Lemma 4.8 requires a common substring of length at least $\lceil \frac{7}{2} \rceil - 1 = 3$, while Property 4.9 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 4.8 is wasteful since the common substrings can start at any position. In contrast, the substrings in Property 4.10 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 4.9 and otherwise Property 4.10. 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

[8]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.

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 invastigating 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.

4.2.2. 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.[9]

Definition 4.11 (*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 4.12 (*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 position of the partitioning of q. Let r'_i be the i-th rotation in $\mathrm{pl}(W)$ 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 $\mathrm{pl}(W)$. 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 $\mathrm{pl}(W)$ (an example is given in Figure 2). Second, the following holds

OBSERVATION 4.13. $lcp(r_j(q), r'_{i+k}) = \min\{lcp(r_j(q), r'_{i+k}), lcp(i + k)\}$

where $lcp(i) = lcp(r'_i, r'_{i+1})$ is an array of the lcps of the adjacent words in $\mathrm{pl}(W)$ precomputed during the initialization of the algorithm. Therefore, we can compute the

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

	...	
	gorham	gorham
	goriszi	zigoris
	gorite	gorite
	gorithimal	algorithim
	gorithm5al	algorithm5
→	gorithma	agorithm
	gorithmal	algorithm
	gorithmeal	algorithme
	gorithsal	algoriths
	goritmal	algoritm
	goriugri	grigoriu
	gorje	jegor
	...	

Fig. 2: Schematic representation of a permuted lexicon. A binary search is performed for the cyclic substring "gorithma" (starting at position 2) of the (misspelled) word "agorithm" (the rotations are given on the left and the original words are given on the right). The common cyclic substrings are underlined.

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 4.9) or less than the length of the current matching piece from q (Property 4.10).

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 4.1.4.

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. [2008; 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.[10]

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.

4.2.3. 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.

4.2.4. 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 4.14. 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 4.15. 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 4.16. 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:

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

— 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 4.17. *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, $\mathrm{WLD}(q, w) = 2$. □

4.2.5. 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.

Example 4.18. 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=`believe` (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 VII). 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.

5. EFFICIENT FUZZY PREFIX MATCHING

In this section we focus on the fuzzy prefix matching problem. The fuzzy prefix matching problem is similar but computationally harder than the fuzzy word matching prob-

Table VII: Average number of words that must be scanned for different longest-common-substring signature lengths (Lemma 4.8) 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

lem, which was the subject of the previous section. 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.

In this section we present two fast and practical off-line algorithms based on the algorithms presented in the previous section: `DeleteMatchPrefix` (Section 5.1) and `PermuteScanPrefix` (Section 5.2). At the end of the section, we propose a simple incremental algorithm and argue that it is sufficiently fast in practice when the prefix is longer than 5 characters (Section 5.3).

5.1. Algorithm: DeleteMatchPrefix

In this section, we first show how the fuzzy word matching algorithm from Section 4.1 can be naturally extended to a fuzzy prefix matching algorithm (Section 5.1.1). We then propose a few optimization techniques that will significantly improve the running time of our algorithm (Sections 5.1.2 and 5.1.3).

5.1.1. The Basic Algorithm. Recall that Lemma 4.3 from Section 4.1 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 4.4 and Lemma 4.5 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 most δ from q, given that $|q| \geq k$. According to Definition 2.2, these are the words from W with PLD at most δ from q, given that $|q| \geq k$.

LEMMA 5.1. *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 2.2), 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 4.4, $\exists s \in U_d(q[k], \delta) \cap U_d(w'[k], \delta)$ with $|s| \geq k - \delta$. □

Example 5.2. 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

Table VIII: 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 5.3, 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

consider. At query time proceed as in Section 4.1, with the difference that PLD instead of WLD is used to verify the candidate matches. Observe that Lemma 4.5 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.

5.1.2. 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 5.3. *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, $\mathrm{PLD}(q, w_2) = t$. If $l = |w_1|$, then $\mathrm{PLD}(q, w_2) < t$ would require $w_1 \preceq w_2$, however, this is not possible since by assumption $w_1 > w_2$. □

Remark 5.4. If the exact value of $\mathrm{WLD}(q, w)$ is not required, then l above could be chosen as the first position in w_1 such that $\mathrm{PLD}(q, w_1[i]) = t$ 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 5.5. 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 5.3 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 $\mathrm{PLD}(q, w_i) = t \leq \delta$ 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 5.3 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 4.2 by computing a lcp array and then using Property 4.13.

5.1.3. 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 4.1.4, by using Lemmas 4.6 and 4.7, 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 VIII 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 5.3 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.

5.2. Algorithm: PermuteScanPrefix

In this section, we introduce a corresponding longest common substring signature for the prefix Levenshtein distance (Section 5.2.1). As before, we formulate the problem as an exact substring matching problem and present an algorithm for its solution (Section 5.2.2). We then propose optimizations to reduce the space usage of the algorithm (Section 5.2.3).

5.2.1. The Longest Common Substring Signature on Prefixes. The key property from Section 4.2 (given in Lemma 4.8) 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 5.6. Consider q=`algor` and w=`alxgorxthmic`. Since $\mathrm{PLD}(q, w) = 1$, according to Lemma 4.8, q should share a substring of length at least 5 with a prefix of w. Note that by using a permuted lexicon from Section 4.2, 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 4.8 on prefixes.

LEMMA 5.7. *If* $\mathrm{PLD}(q, w) \leq \delta$*, 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 4.8 and Definition 2.2 (prefix Levenshtein distance). □

Analogously to Section 4.2, 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.

5.2.2. 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 4.2.2.

Definition 5.8 (*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 5.9 (*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 4.2.2 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 4.13 to compute the substring lengths in constant time and use Lemma 5.3 to skip the verification on adjacent candidate matches that share long prefixes);
(3) Stop the scanning when the condition from Lemma 5.7 is not fulfilled.

5.2.3. 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 4.1 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 5.10. *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.

5.3. 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

Table IX: To compute the prefix Levenshtein distance between the word *algorithm* and the prefix *algro* with threshold $\delta = 1$ incrementally, only the 3 "previous" cells that are at most 1 cell away from the main diagonal are required.

	ϵ	**a**	**l**	**g**	**o**	**r**	**i**	**t**	**h**	**m**
ϵ	0	1								
a	1	0	1							
l		1	0	1						
g			1	0	1					
r				1	1	1				
o					?	?	?			

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.

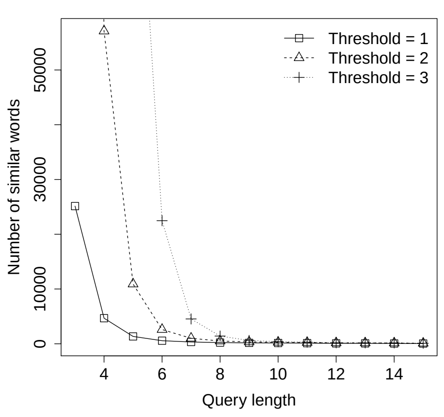
The main idea of the algorithm is as follows. Consider a query $q_i = p_1 \ldots p_i$ and a word w. Consider also the dynamic programming table used to compute the prefix Levenshtein distance between q and w and recall that only the cells that are at most δ cells away from the main diagonal are relevant for the computation. Say $\mathrm{PLD}(q_i, w)$ has been already computed and the dynamic programming table has been already filled. Observe that to compute $\mathrm{PLD}(q_{i+1}, w)$ where $q_i \preceq q_{i+1}$, we only require the $2 \cdot \delta + 1$ non-empty cells from the last row of the dynamic programming table for w (see Table IX). Given a fixed dictionary W, a threshold δ and a query q_i, assume that all $w \in W$ with $\mathrm{PLD}(p_i, w) \leq \delta$ 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 $\mathrm{PLD}(q_{i+1}, w_j)$ 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 5.1.

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 3).

6. EFFICIENT FUZZY SEARCH

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

We start by giving an overview of the baseline algorithm and show how our algorithm addresses its deficiencies. In the next two subsections, we present our new indexing data structures called fuzzy word index (Sections 6.2) and fuzzy prefix index (Section 6.3). We then show how to efficiently solve the fuzzy search problem by using our fuzzy indexes (Section 6.4 and 6.5). In addition, we show how to compute the result incrementally from a previous result by using caching (Section 6.6). We finally show how to efficiently compute S, the set of top-k ranked suggestions for a given query, by making use of the result lists computed by our fuzzy search algorithm (Section 6.7).



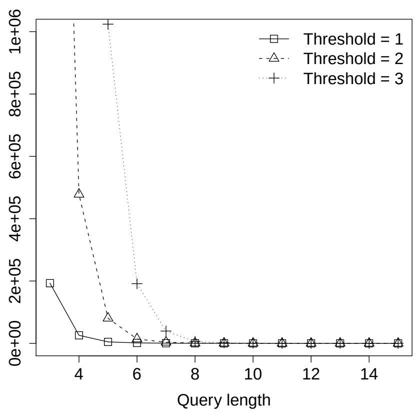


Fig. 3: Average number of matches for different prefix lengths on the dictionaries of the DBLP collection with 600K words (left) and the Wikipedia collection with 28.5M words (right) when using prefix Levenshtein distance with thresholds 1, 2 and 3.

6.1. Overview of Baseline and Our Algorithm

Recall that the fuzzy search problem is as follows. Given a query (consisting of one or more query words), find the list of matching documents D' containing these query words or words similar to them, and at the same time find the set of top ranked query suggestions S. Given a conjuctive query $Q = (q_1, \ldots, q_l)$, recall that $Q_i = \{w \in W \mid \mathrm{LD}(q_i, w) \leq \delta\}$. To compute the list of matching documents D', the fuzzy word (prefix) search defined in Section 2 requires computing the following *intersection of union lists*

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

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 *merging of union lists* where each intersection operator in the formula above is replaced by the union (merge) operator.

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.[11]

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.[12] 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

[11]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.

[12]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 2003].

Fig. 4: A fuzzy inverted list of the set of misspellings of the word *algorithm*. 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.

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

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.

Our algorithm tries to avoid materializing the L_{q_i} lists all along. Instead, our approach is based on precomputed partial fuzzy inverted lists so that at query time each large union list can be represented by a small number of these lists. This addresses (2) and to some extent, also (1). During query processing, the resulting new instance of the intersection of union list problem can then be computed more efficiently by first performing the intersection and then forming the union. This addresses (2).

Here is our query processing algorithm in slightly more detail. Think of a query with $l \geq 2$ keywords and 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 merge these lists by making use of the fact that their lengths are highly skewed; or intersect each of them with the much shorter $R_{q_{l-1}}$ and compute R_{q_l} by merging the obtained lists with total volume much less compared to L_{q_l}.

6.2. 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 6.2.1) and then provide the necessary terminology and definitions (Section 6.2.2). Based on these definitions, we propose and discuss a scheme to construct our new data structure (Section 6.2.3). We then show how to compute a good cover (Section 6.2.4).

6.2.1. 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 4. 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 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 6.1 (*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.

6.2.2. 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 6.2 (*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.

6.2.3. 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:

(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;

Table X: The $\overline{SI}$ and $\overline{SF}$ values (see Definition 6.3) and corresponding percentage of rare words in the collection for different frequency thresholds t computed for the DBLP and the Wikipedia collection (Section 7.1)

	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%

(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 $\mathcal{C}$.

Definition 6.3. 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 X shows the computed $\overline{SI}$ and $\overline{SF}$ values for the DBLP and Wikipedia collections for different values of the frequency threshold t.

LEMMA 6.4. *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 $\{s_w\}_w \in \mathcal{C}$, 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 $Ov(\mathcal{C})$ 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 $\mathcal{C}$. In this regard, it is desirable words with clusters with large $\sum_{w' \in s_w} w'$ to be less affected since they are more expensive to compute. 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 6.5. *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 6.6. 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 6.7 (*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 6.8 (*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 X 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 $s \in \mathcal{C}$ 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 $\mathcal{C}$ is a non-overlapping clustering, then obviously the no-overlap property is always fulfilled. In contrast, if $\mathcal{C}$ 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 $\mathcal{C}$ that do not

Table XI: 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

have a frequent word in common. While this might affect the average cover index, it does not affect its upper bound.

6.2.4. Computing an Optimal Cover. Given a clustering $\mathcal{C}$ 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 a clustering with an upper bound on the average cover index. However, it does not provide a guarantee that the computed cover is optimal for a given keyword q. This is because the scheme considers only a restricted set of clusters, namely those with a representative word w' with $\mathrm{WLD}(q, w') \leq \delta$.

Note that to find an optimal cover by exhaustive search would be too expensive, compared to the total query 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 $s \in \mathcal{C}$ 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 XI shows the average cover index and space overhead achieved on two of our test collections.

6.3. 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 6.3.1) and propose a precomputation algorithm similar to that from Section 6.2.3. We provide evidence why the same algorithm is less effective when applied to prefixes (Section 6.3.2). 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 (Sections 6.3.3 and 6.3.4). As before, at the end of the section we show how to compute a good cover by using our new index (Sections 6.3.5 and 6.3.6).

6.3.1. 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 $|q|$.

6.3.2. 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 $\mathrm{tf}_p = \sum_{w,p \preceq w} \mathrm{tf}_w$, 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

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

We limit the number of clusters assigned to a frequent prefix p' to some small value based on $\mathrm{tf}_{p'}$. An important observation in the clustering algorithm from the previous section was that most of the words in S_q were infrequent. In contrast, S_q may now contain many frequent words if q is relatively short for the following reasons. 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. This makes the transitivity problem more serious than before. Let $\mathrm{PLD}(q, p) \leq \delta$ and $s_p = \{p' \in W_k \mid \mathrm{tf}_{p'} \geq t, \mathrm{PLD}(p, p') \leq \delta\}$. If $|q| \geq |p|$, as described above, it is more likely that $\exists p' \in s_p$ with $\mathrm{PLD}(q, p') > \delta$. If $|q| < |p|$, then too many clusters might be required to cover S_q.

6.3.3. 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 matches prefix with "don't care characters" p up to a prefix, we will write $q \preceq_* p$ .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 6.9. *Given a threshold δ, let p be a prefix with δ "don't care" characters and length k and let $S_p = \{w \in W \mid w[k]$ matches $p\}$. If q is a prefix that matches p, then $\forall w \in S_p, \mathrm{PLD}(q, w) \leq \delta$.*

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

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 relative to p. However, it is not immediately clear how to deal with prefixes that contain deletion or insertion errors.

Example 6.10. 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.

This example shows that the above set of $*$-prefixes is not sufficient to cover all prefixes with PLD within the distance threshold from q. To match prefixes with deletion and/or insertion errors, we use an extended set of $*$-prefixes based on the following lemma

LEMMA 6.11. *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 $\mathrm{PLD}(q,p) \leq \delta$. Then $\cup_{p' \in C} L_{p'}$ covers L_q and $\mathrm{PLD}(q,w) \leq \delta$, 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 type of errors (insertions, deletions and substitutions) as well as their combinations. A detailed version is given in the appendix. □

Remark 6.12. Note that it is not necessary to limit s to $\lceil \delta/2 \rceil$ as in Lemma 6.11, 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.

6.3.4. Refinements. So far, a $*$-prefix $p = p_1 \ldots p_k$ of length k matched all k-prefixes with equal characters on positions $p_i \neq *$. We can in fact extend this set further without breaking the transitivity property. Suppose that p matches all k-prefixes p' with $\mathrm{PLD}(p,p') \leq \delta$. The following lemma shows that the transitivity property is still fulfilled.

LEMMA 6.13. *Let p be a $*$-prefix with s "don't care" characters such that $\mathrm{PLD}(q,p) \leq \delta$. Assume that p' matches p (i.e., $\mathrm{PLD}(p,p') \leq \delta$). Then $\mathrm{PLD}(q,p') \leq \delta$.*

PROOF. Since $\mathrm{PLD}(q,p) \leq \delta$, it must hold that $q \preceq_* p$. Since $\mathrm{PLD}(p,p') \leq \delta$ by assumption and since $q \preceq_* p$, it must hold that $\mathrm{PLD}(q,p') \leq \delta$. □

It is obvious that the new set of matched k-prefixes is a superset of the old set of matched prefixes. Let $p_{i_1}, \ldots, p_{i_{k-s}}$ be the resulting sequence of $k-s$ characters in p after removing the "don't care" characters. Because $\mathrm{PLD}(p, p_{i_1} \ldots p_{i_{k-s}}) \leq \delta$, p will in addition match the k-prefixes staring with $p_{i_1} \ldots p_{i_{k-s}}$. Recall that $*$-prefixes of the type $p_{i_1} \ldots p_{i_{k-s}} *^s$ were required to cover the k-prefixes with insertions errors. These k-prefixes are now covered "for free" by the $*$-prefixes that cover substitution errors (provided they are in W_k^*).

Example 6.14. Assume $k = 4$ and $\delta = 1$. Consider the prefix q=`atxo` (obtained from the word `atom` via an insertion error at position 3). According to Lemma 6.11, the word `atom` will be covered by the $*$-prefix `ato*`. However, since `atom` will now be assigned to `at*o`, which in turn is required to cover substitution errors on position 3, `ato*` is not required anymore.

It was already mentioned that when dealing with prefixes of fixed length, a deletion error in q is always paired with an insertion error. Recall the example with deletion errors from above, where $k = 6$, $\delta = 1$, q=`algoit` (obtained from `algori` by a deletion at positions 5) and $q \notin W_k$. Again, the problem is that q is not within distance threshold from any $*$-prefix $p \in W_k^*$ (e.g., $\mathrm{PLD}(\texttt{algoit}, \texttt{algo*i}) = 2 > \delta$). Lemma 6.11 addresses

Table XII: 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

this by including certain $*$-prefixes from $W^*_{k+\delta}$. An alternative approach that does not involve an additional set of $*$-prefixes is to truncate q to $k - \delta$ characters and consider the existing prefixes $p \in W^*_k$ with $\text{WLD}(q[k-\delta], p) \leq \delta$ (e.g., $\text{WLD}(\texttt{algoi}, \texttt{algo*i}) = 1 \leq \delta$). While this saves space, it has the disadvantage of matching certain words $w \in W$ with $p \preceq_* w$ and $\text{PLD}(q, w) > \delta$ (e.g., $\texttt{algo*i} \preceq_* \texttt{algorihtm}$, but $\text{PLD}(\texttt{algoit}, \texttt{algorihtm}) = 2 > \delta$).

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 6.11 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 more prefixes with PLD within δ from q, however, they have longer fuzzy inverted lists and thus require more space when included in the index.

As before, to limit the size of the index, we 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.

6.3.5. Computing an Optimal Cover. Given a keyword q, a cover computed according to Lemma 6.11 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 $\text{PLD}(q, p) \leq \delta$ 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. Best results in practice were achieved by using a combination of the two methods discussed so far, that is, $*$-prefixes were used to cover the frequent prefixes and the clusters from the beginning of the section to cover the infrequent prefixes. For both methods we employed a greedy heuristic similar to that from Section 6.2.4.

Table XIII and Table XIV 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 XII 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.

6.3.6. Computing a Cover for an Arbitrary Keyword Length. If the new keyword is obtained by adding a letter to the old keyword, in Section 6.4.2 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

Table XIII: 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

Table XIV: 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

cover C by using the $*$-prefixes of length k and then filter C to produce a refined cover C' that fulfills the transitivity property. 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 and then compute L'_i by iterating through L_i and appending the postings with word-ids in S_q by performing a hash look-up. The lists $L'_1, \ldots, L'_m$ are then the refined cover for q. We note that filtering C takes time negligible compared to the total query processing time.

6.4. 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 6.4.1). We then show how the intersection of union lists can be computed more efficiently depending on the particular problem instance (Section 6.4.2).

Recall that L_{q_i} is the fuzzy inverted list of the i-th keyword given as a union list in Equation 2. 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

$$R_{q_{l-1}} \cap \left(\bigcup_{j=1}^{n_l} L_{q_l}^j \right) \tag{3}$$

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

6.4.1. 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.

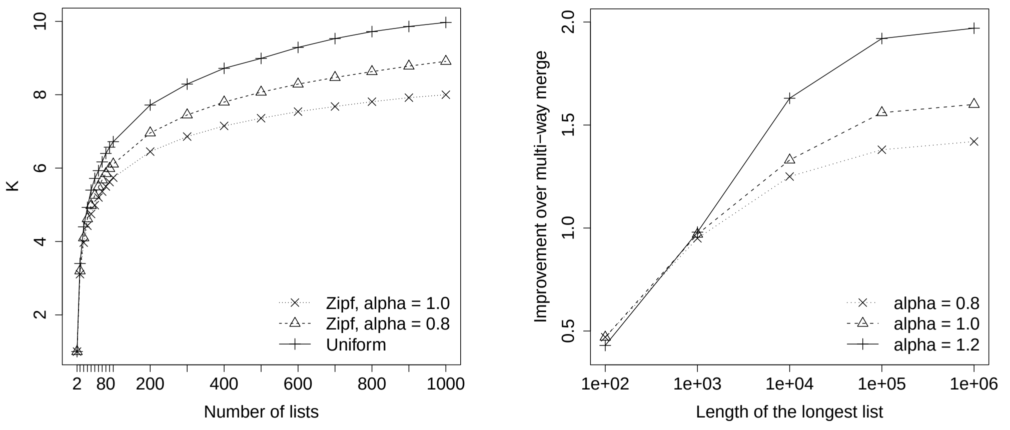


Fig. 5: Left: The K value (defined in Section 6.4.1) for uniform and Zipfian distribution. Right: Advantage of optimal binary merge over multi-way merge on lists with lengths exhibiting various Zipfian distributions given as a ratio of their running times.

Example 6.15. 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 (left) compares the empirically computed K values for the Zipfian and the uniform distribution.

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 (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.

6.4.2. 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 4 (for brevity, we will refer to this as *variant 1* of processing) or by computing its right-hand side (*variant 2* of processing):

$$R_{q_{l-1}} \cap \left(\bigcup_{j=1}^{n_l} L_{q_l}^{j} \right) = \bigcup_{j=1}^{n_l} \left(R_{q_{l-1}} \cap L_{q_l}^{j} \right) \tag{4}$$

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, 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 $L_{q_l}^{i}$ and M the length of R_{q_l}. Regardless of which variant we compute, the short lists in $L_{q_l}^{j}$, $j = 1 \ldots n_l$, are always merged by using a multi-way merge. Let n_l' be the number of remaining lists after performing multi-way merge and 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 by using optimal binary merge. The running time of variant 1 then is

$$c_M \cdot K \cdot n_1 + c_I \cdot (n_1 + n_2) \tag{5}$$

while the running time of variant 2 is

$$c_I \cdot \sum_{i=1}^{n_l'} (n_1 + n_{2,i}) + c_M \cdot k \cdot M \tag{6}$$

$$= c_I \cdot n_l' \cdot n_1 + c_I \cdot n_2 + c_M \cdot k \cdot M \tag{7}$$

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 and 6, we get that variant 2 is faster than variant 1 as long as

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

where $K \propto \log n_l'$ (see Figure 5, left). First, note that $c_I < c_M$. Second, 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 $L_{q_{i-1}}$, n_1 is typically much smaller than n_2 for $l \geq 2$. Since n_l' is always small, the above inequality is typically satisfied. Hence, in most of the cases, only the shortest union list has to be materialized. This results in large advantage of variant 2 over variant 1 when the difference among the union list sizes is large (see Figure 6, left). If the inequality is not satisfied, then we resort to variant 1 of processing.

An alternative way to process the short lists other than multi-way merge is to intersect them with $R_{q_{l-1}}$ by using intersection based on binary searches. The optimal

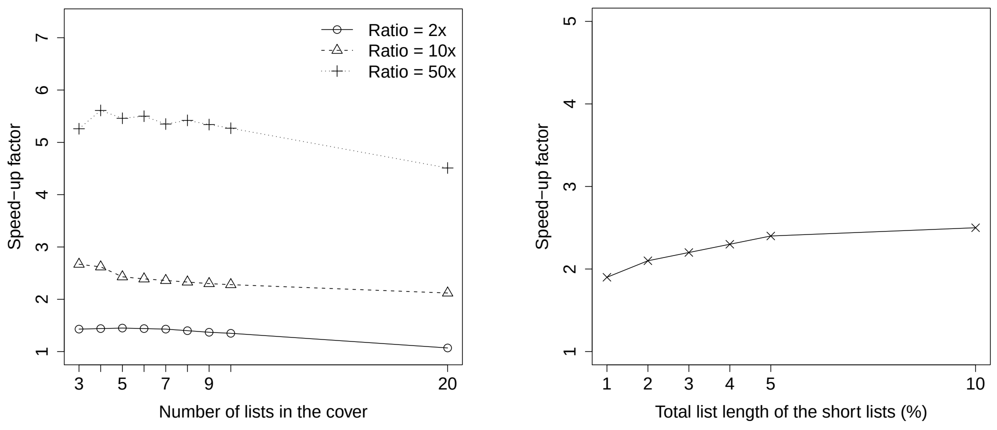


Fig. 6: Left: Improvement of variant 2 over variant 1 of processing the intersection of union lists problem for different number of lists and different ratios between the list sizes (see Section 6.4.2). Right: Advantage of multi-way merge over list intersection based on binary searchers to process the short lists in a cover.

algorithm based on binary searches runs in time $O(n_1 \cdot \log_2 n_2/n_1)$, where n_1 is the shorter and n_2 the longer list. Figure 6 (right) shows the ratio between the average times to process the short lists by using list intersection based on binary searches and by using multi-way merge. The total length of the short lists is given as a percentage of $|R_{q_{l-1}}|$.

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_i^o and C_i^b 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

$$\frac{\sum_{i=1}^{l} C_i^b}{\sum_{i=1}^{l} C_i^o} \tag{8}$$

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) with a union list. Hence, (8) increases with l.

6.5. 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 $L^j_{q_i}$, $j = 1 \dots n_i$ 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 6.4.1. 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.

6.6. 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:

(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 6.3.6;
(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).

6.7. Computing Query Suggestions

So far we solved the first (and biggest) part of the problem stated in Definition 2.5: 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 the following, we first define a score of a query suggestion (Section 6.7.1) and then propose an algorithm to compute a ranked list of suggestions that takes only a small fraction of the total query processing time (Section 6.7.2). The algorithm is relatively straightforward, but to achieve the desired speed, careful algorithm engineering is required. We then show how to incrementally compute a ranked list of suggestions when the user types the query letter by letter by using caching (Section 6.7.3).

6.7.1. 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 2.5. For each suggestion $Q' \in \mathcal{Q}$, we define the following score for Q' with respect to Q:

$$\mathrm{score}(Q', Q) = H(Q') \cdot \mathrm{sim}(Q', Q), \tag{9}$$

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 (9) 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 $Q' = (q'_1, \ldots, q'_l)$. 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 9 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

$$1 - \frac{1}{l} \cdot \sum_{i=1}^{l} \mathrm{LD}(q_i, q'_i) \tag{10}$$

where LD is the word or prefix Levenshtein distance as defined in Section 2.2. 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 (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. 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 $\mathrm{score}(d, Q') = 1$ 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 9 can be interpreted as an instance of the *noisy channel model* [Brill and Moore 2000], where, by using Bayes' theorem,

$$\Pr(Q' \mid Q) \propto \Pr(Q') \cdot \Pr(Q \mid Q') \tag{11}$$

In (11), 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 $\mathrm{score}(Q', Q)$. 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 $\mathrm{sim}(Q', Q)$. Note that there are more sophisticated models to estimate $\mathrm{sim}(Q', Q)$ based on query logs; for example see Li et al. [2006].

6.7.2. Score Computation. Let Q' be a candidate query suggestion from $\mathcal{Q}$. 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 $R(D(\mathcal{Q}), Q_l)$, where $D(\mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} D(Q)$ 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 $R(D(\mathcal{Q}), Q_i)$ not only for $i = l$ but for all $i = 1, \ldots, l$. Then, since $D(Q') \subseteq D(\mathcal{Q})$ 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 $R(D(\mathcal{Q}), Q_i)$ 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 $R(D(\mathcal{Q}), Q_i)$ would be to compute the intersection of $R(D(\mathcal{Q}), Q_l)$, 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 $\mathcal{Q}_i = Q_1 \times \ldots \times Q_i$ 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 $R(D(\mathcal{Q}_i), Q_i)$, 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 $Q' \in \mathcal{Q}$ as follows

(1) Intersect $R(D(\mathcal{Q}), Q_l)$ with $R(D(\mathcal{Q}_i), Q_i)$ to obtain $R(D(\mathcal{Q}), Q_i)$, for $i = 1, \ldots, l-1$. Note that the lists $R(D(\mathcal{Q}_i), Q_i)$ contain the same set of documents with different word occurrences;
(2) Simultaneously traverse all the $R(D(\mathcal{Q}), Q_i)$, 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 $\mathrm{W}(d, Q') = 1$);
(3) Compute a weight $\mathrm{W}(Q', d)$ based on the score of d in D'. If phrases are preferred, then multiply $\mathrm{W}(Q', d)$ 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 $\text{score}(Q', d)$ by $\mathrm{W}(Q', d)$ and update the $H(Q')$'s.
(4) After the lists $R(D(\mathcal{Q}), Q_i)$ 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

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

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

from D in L_{q_i}. The expected length of the result list $R(D(\mathcal{Q}), Q_l)$ 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 $O(N \cdot (p \cdot m)^l)$. 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. Table XV shows a break-down of the total running time in 3 different categories by using queries with two keywords.

6.7.3. 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 $(Q'_i$, $H(Q'_i)$, $\mathrm{sim}(Q, Q'_i))$, 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 $q_l \preceq q''_l$, 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 $\mathrm{LD}(q''_l, q'_l) > \delta$, then obviously Q' cannot be a suggestion for Q'' and it is removed from the list. Suppose $\mathrm{LD}(q''_l, q'_l) \leq \delta$. 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 $\mathrm{sim}(Q'', Q'_i)$ for each triple $(Q'_i$, $H(Q'_i)$, $\mathrm{sim}(Q, Q'_i))$ and sort the new list of triples again with respect to $\mathrm{score}(Q'_i, Q'')$.

Table XVI: 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

7. EXPERIMENTS

In this section, we present the experimental results for all four problems considered in this work. We start by giving overview of our test collections (Section 7.1) and then present the experimental results for fuzzy word matching (Section 7.2), fuzzy prefix matching (Section 7.3), fuzzy keyword-based search (Section 7.4) and fuzzy prefix search (Section 7.5).

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.

7.1. Test Collections

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

(1) **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;

(2) **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 (after cleaning out the dictionary for obvious garbage, its size reduced to around 9M distinct words);

(3) **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 (after cleaning out the dictionary for obvious garbage, its size reduced to around 11M distinct words). The goal was to investigate how well our fuzzy search scales on a larger collection.

7.2. Fuzzy Word Matching

We tested our algorithms on the word dictionaries of two of our test corpora, namely the DBLP corpus and the Wikipedia corpus. 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.

7.2.1. Compared Algorithms. We compared the following algorithms:

(1) `DeleteMatch` (Section 4.1), an algorithm based on truncated deletion neighborhoods combined with unigram frequency filtering. The truncation length parameter (k) was set to 7 characters;
(2) `PermuteScan` (Section 4.2), a sequence-filtering algorithm that combines the longest common substring signature with the pattern partitioning signature with index based on permuted lexicons. We set a recall of 95%.
(3) `DivideSkip` (Section 3), 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 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 out own sequence-filtering algorithm. The inverted lists were compressed by using variable-byte coding;
(4) `msFilter` (Section 3), trie-based algorithms that uses an additional trie built over the reversed strings. It combines tries with pattern partitioning and Levenshtein 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.[13]
(5) `PrefixTrie` (Section 3) 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 7.3.

7.2.2. Queries. To ensure that the performance of the algorithms is not influenced significantly by the query generation method, 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.

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

Table XVII: 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% or 200%	267% or 93%
PermuteScan	500%*	500%
DivideSkip	450%	350%
msFilter	687%	550%

Table XVIII: 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

7.2.3. Discussion. The results are shown in Table XVIII.[14] `DeleteMatch` was the fastest algorithm in general, with `PermuteScan` being a competitive option when δ is dynamic and 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 $\delta = 2$. On the other hand, `msFilter` does not support a dynamic threshold and requires the alphabet specified in advance (the implementation that we tested operated on a basic alphabet consisting of the characters `a...z,0...9`). The large advantage of `PermuteScan` over `DivideSkip` diminishes for $\delta \geq 2$. `PermuteScan` did not perform well for $\delta = 3$ since the effectiveness of its filters 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.

Tale XVII 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 a simple implementation that does not employ any 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. A variant of this algorithm implemented in Boytsov [2011] that does not employ deletion neighborhoods for $\delta = 1$, required around 300% of the vocabulary size.

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

Table XIX: 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%*

7.3. Fuzzy Prefix Matching

In this section, we evaluate our fuzzy prefix matching algorithms. The algorithms were tested under the same experimental setup and on the same word dictionaries from the previous section.

7.3.1. Algorithms Compared. We implemented and compared the following algorithms:

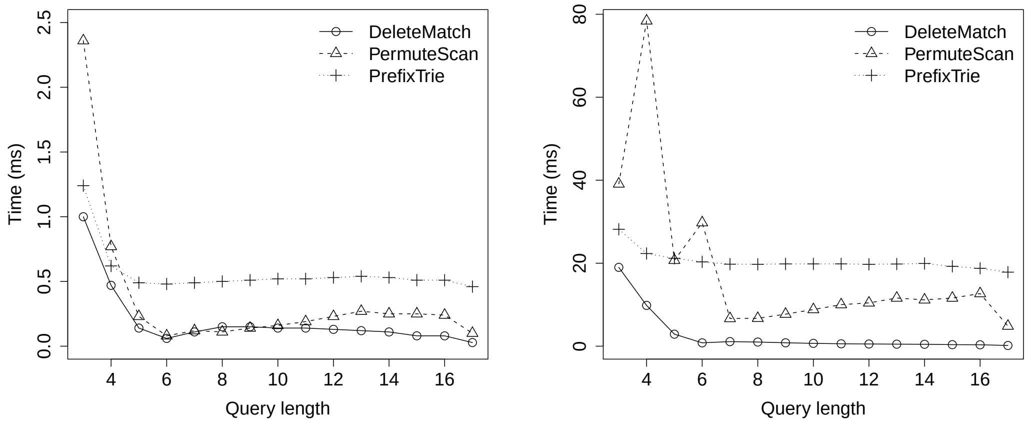
(1) `DeleteMatchPrefix` (Section 5.1), an algorithm based on the longest common subsequence signature and a combination of filters. The truncation length parameter (k) was set to 6 characters;
(2) `PermuteScanPrefix` (Section 5.2), an algorithm based on the longest common substring signature and a combination of filters. The truncation length parameter (k) was set to 8 characters;
(3) `PrefixTrie` (Section 3), the incremental algorithm from Ji et al. [2009] based on a prefix-trie.

7.3.2. Queries. To evaluate the efficiency of our algorithms, we generated 1000 keywords per test collection as in the previous section, 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.

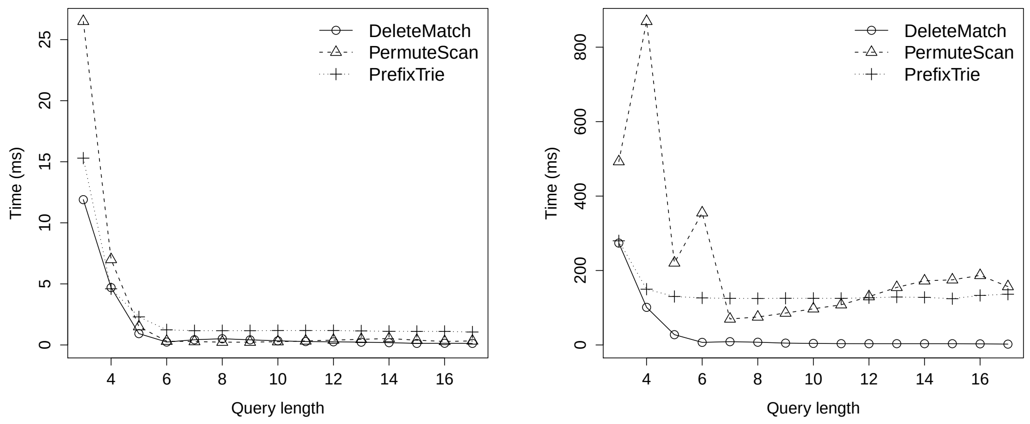
7.3.3. Discussion. Figure 7 shows the average running times for different prefix lengths and different thresholds on two of our test collections. In summary, `DeleteMatchPrefix` was the fastest algorithm, with `PermuteScanPrefix` occasionally being a slightly faster option when $\delta = 1$. The running 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 8 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 5.3 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 does well in practice.

Figure XIX 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.

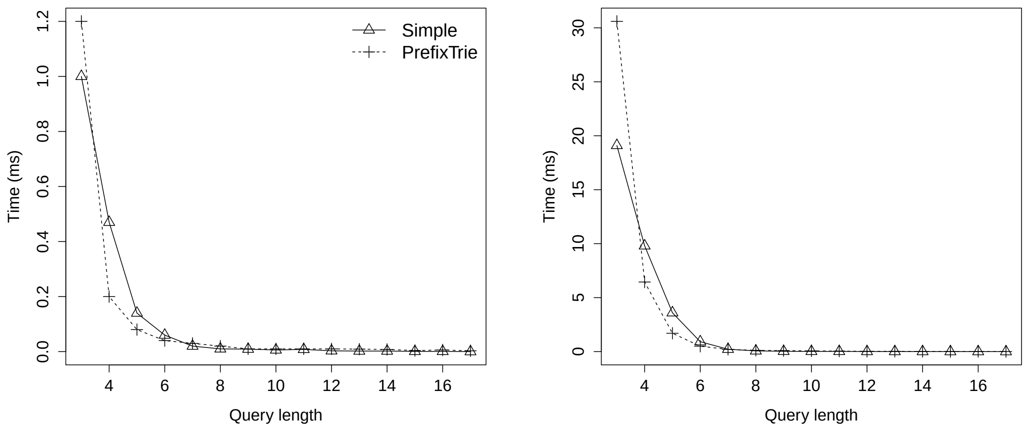


(a) Average running times per prefix length on the dictionary of the DBLP collection for $\delta = 1$ (left) and $\delta = 2$ (right).

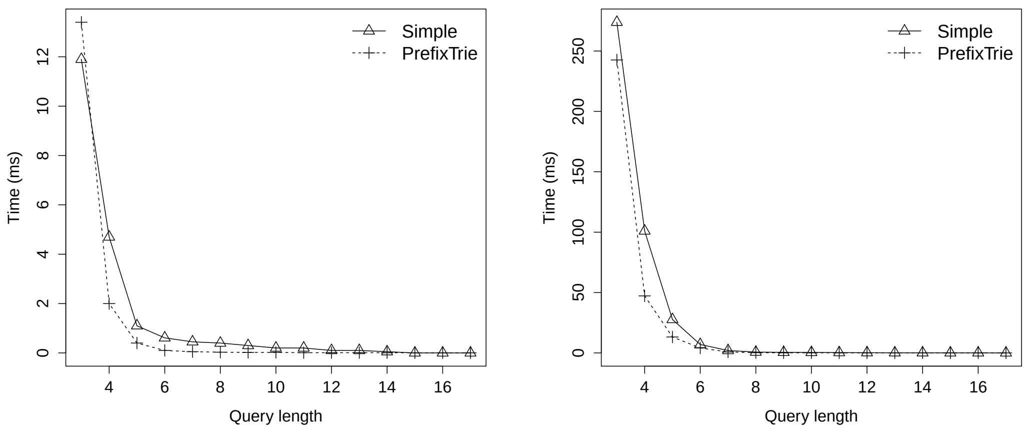


(b) Average running time per prefix length on the dictionary of the Wikipedia collection for $\delta = 1$ (left) and $\delta = 2$ (right).

Fig. 7: 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).

Fig. 8: Average fuzzy prefix matching running times by using an incremental computation of similar prefixes (when the keyword is being typed letter-by-letter, starting from length 3).

7.4. 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.

7.4.1. Algorithms Compared. We evaluated the following algorithms:

(1) The `Baseline` algorithm given in Section 6.1. `Baseline` does not require additional space because employs only an inverted index;
(2) Our proposed method based on a fuzzy word index (Sections 6.2 and 6.4). 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 3. 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 6.4). Also, we insured that the inverted lists are laid out on disk and processed in lexicographic order to minimize disk I/O. 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 4.1) 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.

7.4.2. 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 7.2.2. 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).

7.4.3. Discussion. Table XX 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 XXI shows the average query processing time when the index resides on disk.[15] The time required to decompress the fuzzy inverted lists required around 8%

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

Table XX: 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	110 ms	316 ms	920 ms	2,088 ms
Ours	1.6 ms	23 ms	58 ms	136 ms	399 ms

Table XXI: 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

of the total running time on DBLP and below 5% on Wikipedia and GOV2. The following is a summary of the results:

— Our algorithm improves `Baseline` up to factor of 6;
— 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 and decompressing 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 XXI 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 XXII 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).

Table XXII: 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%

7.5. 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.

7.5.1. 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 6.3) by using $*$-prefixes of length 4, with a single "don't care" character ($k = 4$ and $s = 1$ in Lemma 6.11). 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 5.1 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.

7.5.2. 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 7.2.2. 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.

7.5.3. Discussion. Table XXIII shows the average running time to compute the intersection of union lists on two-word queries and Table XXIV 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 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 increased when the number of query words was large (as predicted at the end of Section 6.4) or when the disk was 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 when fuzzy prefix search was employed compared to fuzzy keyword search, however it remained slow on larger collections for

Table XXIII: 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 ms	41 ms	173 ms	251 ms	1,003 ms

Table XXIV: 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

Table XXV: 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%

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 XXV 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.

Figure 9 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 10 contrasts the average query processing time for different keyword lengths when the intersection is performed with and without caching (see Section 6.6). Obviously, caching reduces the average running time dramatically since the list intersection is done incrementally, by using previous results.

Table XXV 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.

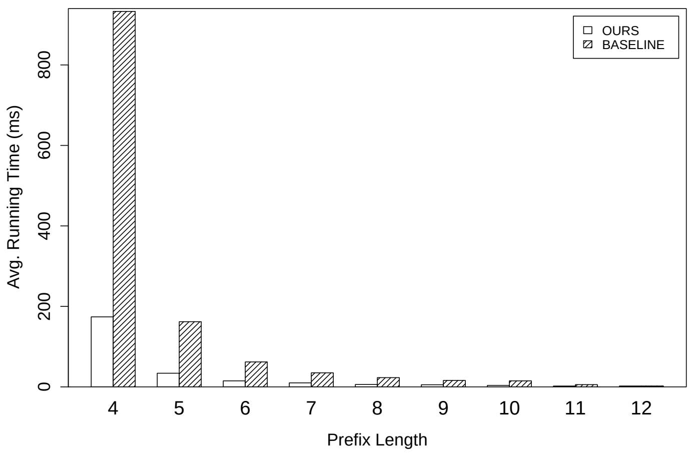


Fig. 9: Average query processing times for different prefix lengths (without using caching) on the Wikipedia collection when the index is in memory.

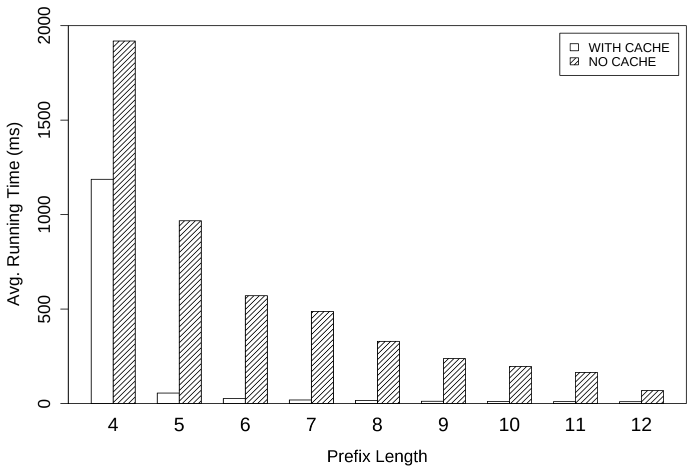


Fig. 10: Average query processing times for different prefix lengths with and without using caching on the Wikipedia collection when the (positional) index is on disk.

8. CONCLUSIONS

We have presented new algorithms for fuzzy word matching and search, with respect to both ordinary Levenshtein distance and prefix Levenshtein distance. Our approach allows fuzzy type-ahead search on each and not only on the last query word as in other approaches. Our algorithms are significantly more efficient and scale to larger collections compared to previous methods. If the respective indexes reside in memory, the improvement is up to a factor of 7. In general, the improvement increases with the number of query words. For large collections, like TREC GOV2, existing methods either yield infeasible index sizes or unacceptably slow query times or both. Our new methods also permit query suggestions based on the contents of the document collection instead of on pre-compiled lists, as in most previous work.

One obvious direction for future work would be to further improve the performance of our algorithms. However, our extensive work on the topic has led us to the conclusion that (i) fuzzy search on very large text collections is a (surprisingly) hard problem, and that (ii) the algorithms we have proposed in this paper come close to what can optimally be achieved when the index resides in memory, at least for the (rather strict) versions of the problems we considered. On the other hand, when the index resides on disk, we believe that a promising research direction is an approach that does not require reading the full fuzzy inverted list for each keyword which is currently the bottleneck of our algorithm.

An interesting approach that adopts existing top-k techniques for ranked retrieval (disjunctive queries) by using the fuzzy search from Ji et al. [2009] has been proposed in Li et al. [2012]. Another direction for further work is to combine our indexes and algorithms with techniques similar to those proposed in this work.

A practically important extension would be to consider similarity measures beyond Levenshtein. For example, certain applications call for a generalized edit distance, with a user-defined error matrix for each pair of letters, or even for arbitrary substrings. For example, an application might consider a “sch“ to be very similar to an “s“, but an “x“ to be very different from a “u“. Note that fuzzy search on UTF-8-encoded strings (which is quite a hassle implementation-wise, but supported by our code because of its practical importance) is a special case of generalized edit distance.

Another practically important issue is that certain phrases are sometimes written as one word, and sometimes as multiple words (e.g. *memorystick* vs. *memory stick*). It seems reasonable to simply not distinguish between such variants. This is easy to achieve in the absence of spelling mistakes: simply index such phrases under both the one-word and the separate-words variant. Making this feature error-tolerant is a challenge though. 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 in the precomputation that *memory stik* is a variant of the common phrase *memory stick*. Solving either of these two tasks efficiently is not easy, especially for phrases with more than two words.

APPENDIX

A. PROOFS OMITTED FOR BREVITY

PROOF OF LEMMA 4.3. 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 $\max\{|w_1|, |w_2|\} - \delta$. $\square$

PROOF OF LEMMA 4.4. 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 $p_1p_2 \dots p_l$ and $s_1s_2 \dots s_l$ such that $p_i \to s_i$ (or equivalently $s_i \to p_i$) 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 $\sum_{i=1}^{j} c(p_i, s_i) \le \delta$ for any $1 \le j \le l$, $\mathrm{WLD}(w_1[k], w_2[k]) > \delta$ implies that $w_1[k]$ and $w_2[k]$ cannot be represented as $p_1 \dots p_i$ and $s_1 \dots s_i$ 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 $t \le \delta$ characters from p_i obtaining $\mathrm{WLD}(w_1[k-t], w_2[k]) \le \delta$. This means there is a transformation $s_t \to p_t$, $t < i$ such that $p_t = \epsilon$ and $|s_t| = t$. Let w_2' be the resulting string after applying $s_t \to p_t$ to $w_2[k]$. We now have $\mathrm{WLD}(w_1[k-t], w_2') \le \delta - t$. According to Lemma 4.3 we can find a matching subsequence between $w_1[k-t]$ and w_2' by applying at most $\delta - t$ deletions. □

PROOF OF LEMMA 6.4 (SECOND PART). Let $\mathcal{C}$ be a given clustering and let $Ov(\mathcal{C})$ be its index space overhead as given in Definition 6.2:

$$Ov(\mathcal{C}) = \frac{\sum_{w \in W} \mathrm{tf}_w \cdot c_w}{\sum_{w \in W} \mathrm{tf}_w}$$

Observe that due to the Zipf's law, $Ov(\mathcal{C})$ 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 $N = \sum_{\mathrm{tf}_w \ge t} \mathrm{tf}_w$. Then we obtain

$$\begin{aligned} Ov(\mathcal{C}) &\approx \frac{1}{N} \cdot \sum_{\mathrm{tf}_w \ge t} \mathrm{tf}_w \cdot c_w \\ &= \frac{1}{N} \cdot \sum_{i=1}^{f} N \cdot \frac{1}{c \cdot i^\alpha} \cdot c_{w_i} \\ &= \frac{1}{c} \cdot \sum_{i=1}^{f} \frac{1}{i^\alpha} \cdot c_{w_i} \end{aligned}$$

If we assume that the c_{w_i}'s are equally distributed, due to linearity of expectation we obtain

$$\mathbb{E}[Ov(\mathcal{C})] \approx \frac{1}{c} \cdot \sum_{i=1}^{f} \frac{1}{i^\alpha} \cdot \mathbb{E}[c_{w_i}] = \mathbb{E}[c_{w_i}]$$

According to Definition 6.3, the latter is approximately equal to $\overline{SF}$. □

PROOF OF LEMMA 6.11. 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 $\mathrm{PLD}(q, p) \le \delta$ 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 $q = p_1 \ldots p_k$. Observe that $p_1 \ldots p_{k-1}* \in W_k^*$ covers all prefixes with a single insertion error. Similarly, $p = p_1 \ldots p_{k-s}*^s \in W_k^*$ 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 $p' \in \{p'' \in W_{k-s} \mid \mathrm{PLD}(q, p'') \leq \delta\}$. It is easy to see that $\mathrm{PLD}(q, p'*^s) \leq \delta$ since $\mathrm{PLD}(q, p') \leq \delta$;
(ii) *substitution errors:* let $\{i_1, \ldots, i_s\}$ be a set of position in q. Then the $*$-prefix $p = p'_1, \ldots, p'_k$, where $p'_j = *$ if $j \in \{j_1, \ldots, j_s\}$ and $p'_j = p_j$ otherwise, covers all prefixes with at least one substitution error on positions $\{j_1, \ldots, j_s\}$. 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 $\{j_1, \ldots, j_s\}$ in q[16];
(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 $\{j_1, \ldots, j_m\}$. Then the $*$-prefix $p = p'_1, \ldots, p'_{k+m}$, where $p'_j = *$ if $j \in \{j_1, \ldots, j_m\}$ and $j \leq k$ and $p'_j = p_j$ otherwise, covers all prefixes with m deletion errors on positions $\{j_1, \ldots, j_m\}$. 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 $\{j_1, \ldots, j_s\}$, 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 $\mathrm{PLD}(q, p) \leq \delta$;
(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 $\mathrm{PLD}(q, p) = 3 > \delta$. 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. □

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their outstandingly painstaking, competent, and constructive comments.

REFERENCES

BAEZA-YATES, R. 2004. A fast set intersection algorithm for sorted sequences. *Lecture Notes in Computer Science 3109*, 400–408.

BAEZA-YATES, R. AND NAVARRO, G. 1998. Fast approximate string matching in a dictionary. In *In Proceeding of the International Symposium on String Processing and Information Retrieval (SPIRE)*. Springer-Verlag, Heidelberg, Germany, 14–22.

[16]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).

BAEZA-YATES, R. A. AND GONNET, G. H. 1999. 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 1999. IEEE Computer Society, Washington, DC, USA, 16–.

BAEZA-YATES, R. A., HURTADO, C. A., AND MENDOZA, M. 2004. Query recommendation using query logs in search engines. In *International Workshop on Clustering Information over the Web (ClustWeb '04* (2004-12-13). Lecture Notes in Computer Science Series, vol. 3268. Springer, Creete, Greece, 588–596.

BARAGLIA, R., CASTILLO, C., DONATO, D., NARDINI, F. M., PEREGO, R., AND SILVESTRI, F. 2009. Aging effects on query flow graphs for query suggestion. In *Proceeding of the 18th ACM conference on Information and knowledge management (CIKM '09)*. ACM, New York, NY, USA, 1947–1950.

BAST, H. AND WEBER, I. 2007. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *Third Conference on Innovative Data Systems Research (CIDR'07)*. VLDB Endowment, Asilomar, CA, USA, 88–95.

BAYARDO, R. J., MA, Y., AND SRIKANT, R. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*. ACM, New York, NY, USA, 131–140.

BELAZZOUGUI, D. 2009. Faster and space-optimal edit distance "1" dictionary. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching*. CPM '09. Springer-Verlag, Berlin, Heidelberg, 154–167.

BHATIA, S., MAJUMDAR, D., AND MITRA, P. 2011. Query suggestions in the absence of query logs. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information (SIGIR '11)*. ACM, New York, NY, USA, 795–804.

BOLDI, P., BONCHI, F., CASTILLO, C., DONATO, D., AND VIGNA, S. 2009. Query suggestions using query-flow graphs. In *Proceedings of the 2009 workshop on Web Search Click Data (WSCD '09)*. ACM, New York, NY, USA, 56–63.

BOYTSOV, L. 2011. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics 16*, 1.1:1.1–1.1:1.91.

BRATLEY, P. AND CHOUEKA, Y. 1982. Processing truncated terms in document retrieval systems. *Information Processing and Management 18,* 5, 257–266.

BRILL, E. AND MOORE, R. C. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics (ACL '00)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 286–293.

CAO, H., JIANG, D., PEI, J., HE, Q., LIAO, Z., CHEN, E., AND LI, H. 2008. Context-aware query suggestion by mining click-through and session data. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '08)*. ACM, New York, NY, USA, 875–883.

CELIKIK, M. AND BAST, H. 2009. Fast error-tolerant search on very large texts. In *Symposium of Applied Computing (SAC '09)*. ACM, New York, NY, USA, 1724–1731.

CHAUDHURI, S., GANTI, V., AND KAUSHIK, R. 2006. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 5.

CHAUDHURI, S. AND KAUSHIK, R. 2009. Extending autocompletion to tolerate errors. In *Proceedings of the 35th International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 707–718.

CHÁVEZ, E., NAVARRO, G., BAEZA-YATES, R., AND MARROQUÍN, J. L. 2001. Searching in metric spaces. *ACM Computational Surveys 33,* 3, 273–321.

COLE, R., GOTTLIEB, L.-A., AND LEWENSTEIN, M. 2004. 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. ACM, New York, NY, USA, 91–100.

CUCERZAN, S. AND WHITE, R. W. 2007. Query suggestion based on user landing pages. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '07)*. ACM, New York, NY, USA, 875–876.

D'AMORE, R. J. AND MAH, C. P. 1985. One-time complete indexing of text: theory and practice. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*. SIGIR '85. ACM, New York, NY, USA, 155–164.

DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2000. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. SODA '00. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 743–752.

DU, M. W. AND CHANG, S. C. 1994. An approach to designing very fast approximate string matching algorithms. *IEEE Trans. on Knowl. and Data Eng. 6,* 4, 620–633.

FERRAGINA, P. AND VENTURINI, R. 2007. Compressed permuterm index. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. SIGIR '07. ACM, New York, NY, USA, 535–542.

FIGUEROA, K., CHÁVEZ, E., NAVARRO, G., AND PAREDES, R. 2006. 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. Springer, Cala Galdana, Menorca, Spain, 279–290.

GAO, W., NIU, C., NIE, J.-Y., ZHOU, M., HU, J., WONG, K.-F., AND HON, H.-W. 2007. 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)*. ACM, New York, NY, USA, 463–470.

GRAVANO, L., IPEIROTIS, P. G., JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2001. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 491–500.

HEINZ, S. AND ZOBEL, J. 2003. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology 54*, 713–729.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40,* 9, 1098–1101.

JAMES, E. B. AND PARTRIDGE, D. P. 1973. Adaptive correction of program statements. *Communications of the ACM 16,* 1, 27–37.

JI, S., LI, G., LI, C., AND FENG, J. 2009. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, New York, NY, USA, 371–380.

JOKINEN, P. AND UKKONEN, E. 1991. Two algorithms for approximate string matching in static texts. In In Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science, P. Jokinen and E. Ukkonen, Eds. *Lecture Notes in Computer Science 520,* 06, 240–248.

KAHVECI, T. AND SINGH, A. K. 2001. Efficient index structures for string databases. In *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 351–360.

KIM, Y., SEO, J., AND CROFT, W. B. 2011. Automatic boolean query suggestion for professional search. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information (SIGIR '11)*. ACM, New York, NY, USA, 825–834.

LEVENSHTEIN, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10,* 8, 707–710.

LI, C., LU, J., AND LU, Y. 2008. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, Washington, DC, USA, 257–266.

LI, G., WANG, J., LI, C., AND FENG, J. 2012. Supporting efficient top-k queries in type-ahead search. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. SIGIR. ACM, New York, NY, USA, 355–364.

LI, M., ZHANG, Y., ZHU, M., AND ZHOU, M. 2006. 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)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 1025–1032.

LI, W. 1992. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory 38*, 1842–1845.

LUND, C. AND YANNAKAKIS, M. 1993. On the hardness of approximating minimization problems. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC '93)*. ACM, New York, NY, USA, 286–293.

MEI, Q., ZHOU, D., AND CHURCH, K. 2008. Query suggestion using hitting time. In *Proceeding of the 17th ACM conference on Information and knowledge management (CIKM '08)*. ACM, New York, NY, USA, 469–478.

MIHOV, S. AND SCHULZ, K. U. 2004. Fast approximate search in large dictionaries. *Computational Linguistics 30*, 451–477.

MOR, M. AND FRAENKEL, A. S. 1982. A hash code method for detecting and correcting spelling errors. *Communications of the ACM 25,* 12, 935–938.

MUTH, R. AND MANBER, U. 1996. Approximate multiple strings search. In *Combinatorial Pattern Matching (CPM '96)*, D. S. Hirschberg and E. W. Myers, Eds. Lecture Notes in Computer Science Series, vol. 1075. Springer, Laguna Beach, California, USA, 75–86.

MYERS, E. W. 1994. A sublinear algorithm for approximate keyword searching. *Algorithmica V12,* 4, 345–374.

MYERS, G. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM 46*, 1–13.

NAVARRO, G. 2001. A guided tour to approximate string matching. *ACM Computing Surveys 33,* 1, 31–88.

NAVARRO, G., BAEZA-YATES, R., SUTINEN, E., AND TARHIO, J. 2000. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin 24*, 2001.

NAVARRO, G. AND SALMELA, L. 2009. Indexing variable length substrings for exact and approximate matching. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*. SPIRE 2009. Springer-Verlag, Berlin, Heidelberg, 214–221.

NEEDLEMAN, S. B. AND WUNSCH, C. D. 1970. 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.

RUSSO, L. M. S., NAVARRO, G., OLIVEIRA, A. L., AND MORALES, P. 2009. Approximate string matching with compressed indexes. *Algorithms 2,* 3, 1105–1136.

SANKOFF, D. 1972. Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A. 69*, 4–6.

SCHOLER, F., WILLIAMS, H. E., YIANNIS, J., AND ZOBEL, J. 2002. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. SIGIR '02. ACM, New York, NY, USA, 222–229.

SCHULZ, K. U. AND MIHOV, S. 2002. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition 5,* 1, 67–85.

SELLERS, P. H. 1974. On the Theory and Computation of Evolutionary Distances. *SIAM Journal on Applied Mathematics 26,* 4, 787–793.

SHI, F. AND MEFFORD, C. 2005. 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).* IEEE Computer Society, Washington, DC, USA, 70–76.

SONG, Y. AND HE, L.-W. 2010. Optimal rare query suggestion with implicit user feedback. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 901–910.

SUTINEN, E. AND TARHIO, J. 1995. On using q-gram locations in approximate string matching. In *Proceedings of the Third Annual European Symposium on Algorithms (ESA '95)*. Springer-Verlag, London, UK, 327–340.

SUTINEN, E. AND TARHIO, J. 1996. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM '96)*. Springer-Verlag, London, UK, 50–63.

UKKONEN, E. 1983. Algorithms for approximate string matching. *Information and Control 64,* 1-3, 100–118.

UKKONEN, E. 1993. Approximate string-matching over suffix trees. *Lecture Notes in Computer Science 684*, 228242.

VINTSYUK, T. K. 1968. Speech discrimination by dynamic programming. *Cybernetics 4,* 1, 52–57. Russian Kibernetika 4(1):81-88 (1968).

WILLETT, P. AND ANGELL, R. 1983. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management 19,* 4, 255–261.

WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Communications of ACM 35,* 10, 83–91.

XIAO, C., WANG, W., AND LIN, X. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of The VLDB Endowment 1*, 933–944.

XIAO, C., WANG, W., LIN, X., AND YU, J. X. 2008. Efficient similarity joins for near duplicate detection. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)*. ACM, New York, NY, USA, 131–140.

ZOBEL, J. AND DART, P. 1995. Finding approximate matches in large lexicons. *Software Practice and Experience 25,* 3, 331–345.

Received February 2007; revised March 2009; accepted June 2009