# Indexing Methods for Approximate Dictionary Searching: Comparative Analysis

Leonid Boytsov

The primary goal of this paper is to survey state of the art indexing methods for approximate dictionary searching. To improve understanding of the field, we introduce a taxonomy that classifies all methods into direct methods and sequence-based filtering methods. We focus on infrequently updated dictionaries, which are used primarily for retrieval. Therefore, we consider indices that are optimized for retrieval rather than for update. The indices are assumed to be associative, i.e., capable of storing and retrieving auxiliary information, such as string identifiers. All solutions are lossless and guarantee retrieval of strings within a specified edit distance $k$. Benchmark results are presented for the practically important cases of $k = 1, 2, 3$. We concentrate on natural language datasets, which include synthetic English and Russian dictionaries, as well as dictionaries of frequent words extracted from the ClueWeb09 collection. In addition, we carry out experiments with dictionaries containing DNA sequences. The paper is concluded with a discussion of benchmark results and directions for future research.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Non-numerical algorithms and problems—*Pattern matching*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: approximate searching, Levenshtein distance, Damerau-Levenshtein distance, frequency distance, neighborhood generation, $q$-gram, $q$-sample, metric trees, trie, $k$-errata tree, frequency vector trie, agrep, NR-grep.

**ACM Reference Format:**
ACM J. Exp. Algor. 0, 0, Article A ( 00), 92 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

As computers are used in an ever-widening variety of lexical processing tasks, the problem of error detection and correction becomes more critical. Mere volume, if nothing else, will prevent the employment of manual detection and correction procedures.

Damerau [1964]

## 1. INTRODUCTION

Detection and correction of errors is a challenging task, which gave rise to a variety of approximate search algorithms. At present, these algorithms are used in spell-checkers [Zamora et al. 1981; Kukich 1992; Brill and Moore 2000; Wilbur et al. 2006],

Author's address: 11710 Old Georgetown Rd, Unit 612, North Bethesda, MD, 20852, USA; e-mail: leo@boytsov.info.

computer-aided translation systems (e.g., Trados[1]), optical character recognition systems [Ford et al. 2001], spoken-text recognition and retrieval systems [Vintsyuk 1968; Velichko and Zagoruyko 1970; Ng and Zue 2000; Glass 2003], computational biology [Gusfield 1999; Ozturk and Ferhatosmanoglu 2003; Kahveci et al. 2004], phonology [Covington 1996; Kondrak 2003], and for phonetic matching [Zobel and Dart 1996]. Approximate dictionary searching is important to computer security, where it can be used to identify "weak" passwords susceptible to a dictionary attack [Manber and Wu 1994a]. Another important application is a detection of similar domain names that are registered mostly for typo-squatting [Moore and Edelman 2010]. This is also frequently required in disputes over almost identical domain names, particularly when brand names are involved.[2]

Along with other sources of corrupt data, such as recognition of spoken and printed text, a lot of errors are attributable to the human factor, i.e., cognitive or mechanical errors. In particular, English orthography is notorious for its inconsistency and, therefore, poses significant challenges. For instance, one of the most commonly misspelled words is "`definitely`". Though apparently simple, it is often written as "`defin`**`a`**`tly`" [Collins 2009].

An example of a frequent mechanical error is reversal of adjacent characters, known as a transposition. Transpositions account for 2-13 percent of all misspelling errors [Peterson 1986]. Therefore, it is important to consider transposition-aware methods.

Search methods can be classified into *on-line* and *off-line* methods. On-line search methods include algorithms for finding approximate pattern occurrences in a text that cannot be preprocessed and indexed. A well-known on-line search algorithm is based on the dynamic programming approach [Sellers 1974]. It is $O(n \cdot m)$ in time and $O(n)$ in space, where $n$ and $m$ are the lengths of a pattern and a text, respectively. The on-line search problem was extensively studied and a number of algorithms improving the classic solution were suggested, including simulation of non-deterministic finite automata (NFA) [Wu and Manber 1992b; Navarro and Raffinot 2000; Navarro 2001b], simulation of deterministic finite automata (DFA) [Ukkonen 1985b; Melichar 1996; Kurtz 1996; Navarro 1997b], and bit-parallel computation of the dynamic programming matrix [Myers 1999; Hyyrö 2005].

However, the running time of all on-line search methods is proportional to the text size. Given the volume of textual data, this approach is inefficient. The necessity of much faster retrieval tools motivated development of methods that rely on text preprocessing to create a search index. These search methods are known as *off-line* or *indexing* search methods.

There are two types of indexing search methods: *sequence-oriented* methods and *word-oriented* methods. Sequence-oriented methods find *arbitrary substrings* within a specified edit distance. Word-oriented methods operate on a text that consists of strings divided by separators. Unlike sequence-oriented methods, they are designed to find only *complete* strings (within a specified edit distance). As a result, a word-oriented index would fail to find the misspelled string "`wiki pedia`" using "`wikipedia`" as the search pattern (if one error is allowed). In the case of natural languages – on which we focus in this survey – this is a reasonable restriction.

The core element of a word-oriented search method is a *dictionary*. The dictionary is a collection of distinct searchable strings (or string sequences) extracted from the text. Dictionary strings are usually indexed for faster access. A typical dictionary index allows for exact search and, occasionally, for prefix search. In this survey, we review

[1]http://trados.com
[2]A 2006 dispute on domains telefónica.cl and telefonica.cl resulted in court canceling the "accented" domain version, because it was too similar to the original one [Zaliznyak 2006].

associative methods that allow for an *approximate* dictionary search. Given the search pattern $p$ and a maximum allowed edit distance $k$, these methods retrieve all dictionary strings $s$ (as well as associated data) such that the distance between $p$ and $s$ is less than or equal to $k$.

One approach to sequence-oriented searching also relies on dictionary matching. During indexing, all unique text substrings with lengths in a given interval (e.g., from 5 to 10) are identified. For each unique substring $s$, the indexing algorithm compiles the list of positions where $s$ occurs in the text. Finally, all unique substrings are combined into a dictionary, which is used to retrieve occurrence lists efficiently. Although this method is not space efficient, it can be used for retrieval of DNA sequences.

The problem of index-based approximate string searching, though not as well studied as on-line string searching, attracted substantial attention from practitioners and scientists. There are many experimental, open source, and commercial word-oriented indexing applications that support approximate searching: Glimpse [Manber and Wu 1994b], dtSearch[3], to name but a few. Other examples of software that employ approximate dictionary searching are spell-checkers Ispell [Kuenning et al. 1988] and GNU Aspell,[4] as well as spoken-text recognition systems: SUMMIT [Zue et al. 1989; Glass 2003],[5] Sphinx [Siegler et al. 1997],[6] and HTK [Woodland et al. 1994].[7].

Commonly used indexing approaches to approximate dictionary searching include the following:

— *Full and partial neighborhood generation* [Gorin 1971; Mor and Fraenkel 1982; Du and Chang 1994; Myers 1994; Russo and Oliveira 2005];
— *$q$-gram indices* [Angell et al. 1983; Owolabi and McGregor 1988; Jokinen and Ukkonen 1991; Zobel et al. 1993; Gravano et al. 2001; Carterette and Can 2005; Navarro et al. 2005];
— *Prefix trees* (tries) [James and Partridge 1973; Klovstad and Mondshein 1975; Baeza-Yates and Gonnet 1990; Ukkonen 1993; Cole et al. 2004; Mihov and Schulz 2004];
— *Metric space methods* [Baeza-Yates and Navarro 1998; Navarro et al. 2002; Fredriksson 2007; Figueroa and Fredriksson 2007].

There exist several good surveys on this topic [Hall and Dowling 1980; Peterson 1980; Kukich 1992; Owolabi 1996; Navarro 2001a; Navarro et al. 2001; Sung 2008], yet, none addresses the specifics of dictionary searching in sufficient detail. We aim to fill this gap by presenting a taxonomy of state of the art indexing methods for approximate dictionary searching and experimental comparison results.

***Paper organization***. This paper can be divided into three parts: basic concepts, method descriptions, and experiments, which are followed by conclusions and suggestions for future work. Auxiliary material that is not critical for understanding the paper is presented in appendices.

We introduce terminology and formalize the problem in Section 2.1. In Section 2.2 we discuss the basics of the Levenshtein distance and its generalizations. In particular, we discuss the Damerau-Levenshtein distance, which treats transpositions as basic edit operations. We also show that the commonly used formula for evaluating the Damerau-Levenshtein distance does not satisfy the triangle inequality and cannot be used with metric space methods directly. Instead, we note that the Damerau-

[3]In dtSearch documentation this feature is called *fuzzy searching*, see http://dtsearch.com.
[4]http://aspell.net
[5]http://groups.csail.mit.edu/sls/technologies/asr.shtml
[6]http://cmusphinx.sourceforge.net
[7]http://htk.eng.cam.ac.uk

Levenshtein distance should be calculated using the algorithm suggested by Lowrance and Wagner [1975].

Some of the surveyed methods rely on the projection of strings into a general vector space. Then original dictionary queries are substituted with frequency distance queries in the projected space. This step gives the list of candidate strings, which are compared with the pattern element-wise. Because frequency distance queries are seldom discussed in the literature, we explain them in Section 2.3.

Section 2.2 and Section 2.3 can be skipped in a first reading. To understand most of the material, it is sufficient to know that the Levenshtein distance is equal to the minimum number of basic edit operations (i.e., insertions, deletions, and substitutions) needed to transform one string into another. This function is a metric and satisfies the axioms of a metric space. The Levenshtein distance can be lower bounded by the frequency distance.

In Section 3 we outline the taxonomy that classifies all the indexing methods into direct methods and sequence-based filtering methods. In Section 4 we briefly describe exact search algorithms that are used as a part of approximate search methods. In Section 5 we discuss efficient on-line (sequential) search methods and sketch an on-line algorithm that serves as a comparison baseline for all indexing methods in our experiments. In addition, it is embedded in many filtering methods. The description of direct indexing methods is given in Section 6, while the description of indexing methods based on sequence filtering is given in Section 7. We also survey several hybrid algorithms, which are discussed jointly with the other methods.

Experimental results are presented in Section 8. Section 9 concludes the paper.

## 2. BASIC CONCEPTS

### 2.1. Definitions and Assumptions

Let $\Sigma = \{\Sigma_i\}$ be a finite ordered alphabet of the size $|\Sigma|$. A string is a *finite* sequence of characters over $\Sigma$. The set of all strings of the length $n$ over $\Sigma$ is denoted by $\Sigma^n$, while $\Sigma^* = \bigcup_{n=1}^{\infty} \Sigma^n$ represents the set of all strings.

Unless otherwise specified, we use $p$, $s$, $u$, $v$, $w$ to represent arbitrary strings and $a$, $b$, $c$ to represent *single-character* strings, or simply characters. The empty string is represented by $\epsilon$. For any string $s \in \Sigma^*$ its length is denoted by $|s|$. A series of string variables and/or character variables represents their concatenation.

To avoid confusion between string variables with indices and string characters, we denote the $i$-th character of the string $s$ by $s_{[i]}$. A contiguous subsequence of string characters is a *substring*. The substring of $s$ that starts at position $i$ and ends at position $j$ is denoted by $s_{[i:j]}$, i.e., $s_{[i:j]} = s_{[i]}s_{[i+1]} \dots s_{[j]}$. The reversed string, i.e., the string $s_{[n]}s_{[n-1]} \dots s_{[1]}$, is denoted by $\text{rev}(s)$.

Assume that the string $s$ is represented as a concatenation of three possibly empty substrings $s_1$, $s_2$, and $s_3$, i.e., $s = s_1 s_2 s_3$. Then substring $s_1$ is a *prefix* of $s$, while substring $s_3$ is a *suffix* of $s$.

A substring of fixed size $q$ is called $q$*-gram* (also known as *n-gram*). $q$-grams of sizes one, two and three have special names: *unigram*, *bigram*, and *trigram*. Consider a string $s$ of the length $n$. We introduce function $q$-grams$(s)$ that takes the string $s$ and produces a sequence of $n - q + 1$ $q$-grams contained in $s$:

$$q\text{-grams}(s) = s_{[1:q]}, s_{[2:q+1]}, \dots, s_{[n-q+1:n]}.$$

If $n < q$, $q$-grams$(s)$ produces the empty sequence. It can be seen that $q$-grams$(s)$ is essentially a mapping from $\Sigma^*$ to the set of strings over the alphabet $\Sigma^q$, which is comprised of all possible $q$-grams.

Table I: Notation

| **Concept** | **Notation** |
|---|---|
| Alphabet | $\Sigma$ |
| Set of strings of the length $l$ over $\Sigma$ | $\Sigma^l$ |
| Set of all strings over $\Sigma$ | $\Sigma^*$ |
| Strings | $p$ (pattern), $s$, $u$, $v$ , $w$ |
| The empty string | $\epsilon$ |
| String length (or the number of set elements) | $\|s\|$ |
| Single-character strings | $a$, $b$, $c$ |
| The $i$-th character of the string $s$ | $s_{[i]}$ |
| Substring of $s$ that starts at position $i$ and ends at position $j$ | $s_{[i:j]}$ |
| Operation that consists in deletion of the $i$-th character | $\Delta_i(s)$ |
| $q$-gram length | $q$ |
| The sequence of $q$-grams contained in the string $s$ | $q$-grams$(s)$ |
| Reduced alphabet | $\sigma$ |
| Hash function | $h(s)$ |
| Cost of mapping $u$ into $v$ | $\delta(u \to v)$ |
| Set of basic edit operations | $\mathbb{B}$ |
| Edit distance between $s$ and $p$ | ED$(s, p)$ |
| Dictionary | $W$ |
| Number of dictionary strings | $N$ |
| Number of dictionary strings of the length $l$ | $N(l)$ |
| The largest $N(l)$ for $\|l - n\| \le k$ | $L(n, k) = \max\limits_{\|l-n\| \le k} N(l)$ |
| Average length of dictionary strings | $\lambda$ |
| Maximum length of dictionary strings | $\lambda_m$ |
| Maximum allowed edit distance | $k$ |
| Size of the full $k$-neighborhood of $p$ | $U_k(p)$ |
| Vectors | $x$, $y$, $z$ |
| Null (zero) vector | $\vec{0}$ |
| Frequency vector of the string $s$ ($s$ can be also a $q$-gram sequence) | vect$(s)$ |
| Signature of string $s$ ($s$ can be also a $q$-gram sequence) | signat$(s)$ |
| Frequency distance between vectors (or signatures) | FD$(x, y)$ |
| Distance in a general metric space | $d(x, y)$ |
| Maximum allowed distance in a general metric or a vector space | $R$ |
| Probability of event $A$ | Pr(A) |
| Iverson bracket | $[P] = \begin{cases} 1, & \text{if P is true} \\ 0, & \text{otherwise.} \end{cases}$ |

The string $s$ can be transformed into a *frequency vector* (we also use the term *unigram frequency vector*). The frequency vector is a vector of size $|\Sigma|$, where the $i$-th element contains the number of occurrences of the $i$-th alphabet character in the string $s$. The frequency vector produced by the string $s$ is denoted by $\mathrm{vect}(s)$.

Because $q$-grams$(s)$ is itself a string over the alphabet $\Sigma^q$, it can be also transformed into a frequency vector vect($q$-grams$(s)$). To distinguish it from the (unigram) frequency vector obtained directly from the string $s$, we use the term *$q$-gram frequency vector*.

A *signature* is a variant of a frequency vector. It is a binary vector of size $|\Sigma|$, where the $i$-th element is equal to one, if and only if the $i$-th alphabet character belongs to $s$, and is zero otherwise. The signature produced by the string $s$ is denoted by $\mathrm{signat}(s)$.

The notation is summarized in Table I.

Several surveyed methods reduce the dimensionality of the search problem by projecting the original alphabet $\Sigma$ to a smaller alphabet $\sigma$ using a hash function $h(c)$. The alphabet $\sigma$ is called a *reduced alphabet* (see Appendix B for a discussion of hash functions). The hash function $h(c)$ induces a character-wise projection from the set of strings over the original alphabet $\Sigma$ to the set of strings over the reduced alphabet $\sigma$ in a straightforward way. Given a string $s$ of the length $n$, a corresponding projection $h(s)$ is given by:

$$h(s_{[1]}s_{[2]}\ldots s_{[n]}) = h(s_{[1]})h(s_{[2]})\ldots h(s_{[n]}). \tag{1}$$

One of the key concepts in this paper is an *edit distance* $\mathrm{ED}(p, s)$, which is equal to the minimum number of edit operations that transform a string $p$ into a string $s$. A *restricted edit distance* is computed as the minimum number of *non-overlapping* edit operations that make two strings equal (and do not act twice on the same substring). If the edit operations include only insertions, deletions, and substitutions, it is the *Levenshtein distance*. If, in addition, transpositions are included, it is the *Damerau-Levenshtein* distance. The *Hamming distance* is a variant of the edit distance where the only basic edit operations are substitutions.

The restricted Levenshtein distance is always equal to the unrestricted Levenshtein distance, but this is not always the case for the Damerau-Levenshtein distance. We continue the discussion of edit distances (and algorithms to compute them) in Section 2.2.

In the Introduction we define approximate dictionary searching as finding a set of strings that match the pattern within $k$ errors. Now we give formal definitions.

*Definition* 2.1. A *dictionary* $W = (s_1, s_2, \ldots, s_N)$ is an ordered set of strings, where $N$ is the number of dictionary strings.

*Definition* 2.2. Let $p$ be a search pattern (query string) and $k$ be a maximum allowed *restricted* edit distance. Then the problem of approximate dictionary searching consists in finding all indices $i$ in $W = (s_1, s_2, \ldots, s_N)$ such that $\mathrm{ED}(s_i, p) \le k$.

Note that Definition 2.2 employs a *restricted* distance. This definition also implies that we consider *associative* methods that are capable of retrieving both strings and associated data, such as string identifiers. Associated data is also known as *satellite* data.

We use $N(l)$ to denote the number of strings of the length $l$. The average and the maximum length of a dictionary string are denoted by $\lambda$ and $\lambda_m$, respectively. Note that the total number of characters in dictionary strings is equal to $\lambda N$.

*Definition* 2.3. The density of the dictionary is equal to $N$ divided by $\sum_{n=0}^{\lambda_m} |\Sigma|^n$. The latter is the total number of possible strings over the alphabet $\Sigma$ not longer than $\lambda_m$ characters.

We informally call a dictionary *dense*, if its density is close to 1.

Depending on the type of edit distance used, we divide all methods into *transposition-aware* and *transposition-unaware* methods. Transposition-unaware methods use the Levenshtein distance, while transposition-aware methods use the Damerau-Levenshtein distance.

We consider a version of the problem where the dictionary is preprocessed to build an *index*. The index is an auxiliary data structure that allows one to avoid examining every dictionary string at search time and makes retrieval faster. We focus on methods with indices optimized for retrieval rather than updating. However, we prefer methods with indices that are updateable without rebuilding most of the structure.

### 2.2. Edit Distance

The history of edit distance begins with Damerau [1964] who presented misspelling statistics and a method to correct a single misspelling. Damerau focused on single-character insertions, deletions, substitutions, and transpositions, which accounted for the majority of misspelling errors (about 80 percent) in a sample of human keypunched texts. Independently, Levenshtein [1966] proposed a similarity function defined as as the minimum number of insertions, deletions, and substitutions (but not transpositions) required to obtain one string from another.

*2.2.1. Edit Script and Basic Edit Operations.* In a more general perspective, one string can be transformed into another by a sequence of *atomic* substring transformations. This sequence is an *edit script* (also known as a *trace*), while atomic substring transformations are *basic* edit operations. A basic edit operation that consists in mapping string $u$ into string $v$ is represented by $u \to v$ (for simplicity of exposition, we omit specification of exact positions where basic operations are applied). We denote the set of basic edit operations by $\mathbb{B}$.

Basic edit operations are usually restricted to the following set of *single-character* operations:

— Insertion: $\epsilon \to b$;
— Deletion: $a \to \epsilon$;
— Substitution: $a \to b$ (replacement);

In some cases, $\mathbb{B}$ is expanded to include transpositions, which consist in reversal of adjacent characters: $ab \to ba$.

PROPERTY 2.4. *We assume that $\mathbb{B}$ satisfies the following:*

— *If $u \to v \in \mathbb{B}$, then the reverse operation $v \to u$ also belongs to $\mathbb{B}$ (symmetry);*
— *$a \to a \in \mathbb{B}$ (single-character identity operations belong to $\mathbb{B}$);*
— *$\mathbb{B}$ is* complete*: for any two strings $p$ and $s$ there always exists an edit script that transforms $p$ into $s$.*

Note that $\mathbb{B}$ is not necessarily finite.

*2.2.2. Definition of Edit Distance.* Similarity of two strings can be expressed through the length of an edit script that makes strings equal:

*Definition* 2.5. Given a set of basic edit operations, the edit distance $\mathrm{ED}(p, s)$ is equal to the length of a shortest edit script that transforms string $p$ into string $s$. A shortest script that transforms $p$ into $s$ is an *optimal* edit script. If the set of basic edit operations contains only insertions, deletions, and substitutions, it is the Levenshtein distance (an identity operation is a special case of substitution). If, in addition, the set of basic edit operations includes transpositions, it is the Damerau-Levenshtein distance.

The edit distance can be interpreted as the minimum cost at which one string can be transformed into another. It can be generalized in two ways. First, basic edit operations can be assigned individual costs $\delta(a \to b)$ [Wagner and Fischer 1974]. We extend the cost function $\delta()$ to an edit script $E = a_1 \to b_1, a_2 \to b_2, \ldots, a_{|E|} \to b_{|E|}$ by defining $\delta(E) = \sum_{i=1}^{|E|} \delta(a_i \to b_i)$. We now let the distance from a string $p$ to a string $s$ be the minimum cost of all edit scripts that transform $p$ into $s$. This edit distance variant is commonly referred to as a *generalized Levenshtein distance*.

Second, the set of basic edit operations $\mathbb{B}$ can be expanded to allow weighted substitutions of arbitrary strings rather than single-character edit operations [Ukkonen 1985a; Veronis 1988]. This variant is termed as an *extended edit distance*. For example, $\mathbb{B}$ may contain a unit cost operation $\mathtt{x} \to \mathtt{ks}$. Then the extended edit distance between strings “taxi” and “taksi” is equal to *one*, whereas the regular Damerau-Levenshtein distance is equal to *two*. It is also possible to make cost function $\delta()$ conditional on substring positions [Brill and Moore 2000].

*Definition* 2.6. Given a set of basic edit operations $\mathbb{B}$ and a function $\delta()$, which assign costs to all basic edit operations from $\mathbb{B}$, the generic edit distance between strings $p$ and $s$ is defined as the minimum cost of an edit script that transforms $p$ into $s$.

PROPERTY 2.7. *We assume that the cost function $\delta(u \to v)$ satisfies the following:*

— $\delta(u \to v) \in \mathbb{R}$ *(the cost function is real valued)*
— $\delta(u \to v) = \delta(v \to u)$ *(symmetry)*
— $\delta(u \to v) \geq 0$*,* $\delta(u \to u) = 0$*, and* $\delta(u \to v) = 0 \;\Rightarrow\; u = v$ *(positive definiteness)*
— $\forall \gamma > 0$ *the set of basic operations* $\{u \to v \in \mathbb{B} \,|\, \delta(u \to v) < \gamma\}$ *is finite (finiteness of a subset of basic edit operations whose costs are bounded from above)*

Note that the last property holds automatically for finite $\mathbb{B}$.

THEOREM 2.8. *From Properties 2.4 and 2.7 it follows that*

— *For any two strings $p$ and $s$, there exists a script with the minimum cost, i.e., the edit distance from $p$ to $s$ is properly defined.*
— *The generic edit distance described by Definition 2.6 is a metric [Wagner and Fischer 1974].*

The proof is outlined in Appendix D.1.

The edit distance is a metric even if the cost function $\delta()$ is not subadditive. Moreover, because a sequence of overlapping operations that transforms $u$ into $v$ may have a smaller cost than $\delta(u \to v)$, $\delta(u \to v)$ may be greater than $\mathrm{ED}(u, v)$. Consider, for example, the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ where the symmetric and *non-subadditive* $\delta()$ is defined in the following way:

$$\begin{gathered}\delta(\mathtt{a} \to \mathtt{c}) = \delta(\mathtt{b} \to \mathtt{c}) = 1 \\ \delta(\mathtt{a} \to \epsilon) = \delta(\mathtt{b} \to \epsilon) = \delta(\mathtt{c} \to \epsilon) = 2 \\ \delta(\mathtt{a} \to \mathtt{b}) = 3.\end{gathered}$$

It can be seen that $3 = \delta(\mathtt{a} \to \mathtt{b}) > \delta(\mathtt{a}, \mathtt{c}) + \delta(\mathtt{c}, \mathtt{b}) = 2$. That is the optimal edit script $(\mathtt{a} \to \mathtt{c},\ \mathtt{c} \to \mathtt{b})$ transforms “a” into “b” at cost two.

*2.2.3. Restricted Edit Distance.* Subadditivity of edit distance makes it possible to use edit distance with metric space methods, such as the Burkhard-Keller tree (BKT) [Burkhard and Keller 1973]. Nonetheless, a problem of minimization over the set of all possibly overlapping edit operations may be hard. To offset computational complexities, a similarity function defined as the minimum cost of a *restricted edit script* is commonly used. The restricted edit script does not contain overlapping edit operations and does not modify a single substring twice. We refer to corresponding edit distance as the *restricted edit distance*.

OBSERVATION 2.9. *Any unrestricted edit distance is a lower bound for the corresponding restricted edit distance.*

OBSERVATION 2.10. *The restricted unit-cost Levenshtein distance is equal to the unrestricted unit-cost Levenshtein distance.*

A proof immediately follows from the observation that an optimal edit script contains single-character deletions, insertions, or substitutions that never modify a character twice.

OBSERVATION 2.11. *The unrestricted Damerau-Levenshtein distance and the restricted Damerau-Levenshtein distance are different functions. Furthermore, the restricted Damerau-Levenshtein distance is not a metric, because it is not subadditive.*

PROOF. Damerau-Levenshtein distance treats transposition (i.e., two-character reversal of adjacent characters) as a basic edit operation. To prove the claim, we consider an example where prohibition on modification of already transposed characters distinguishes restricted and the unrestricted Damerau-Levenshtein distance. □

Let us consider strings "ab", "ba", and "acb". On one hand, the shortest *unrestricted* edit script that transforms "ba" into "acb" ($\mathtt{ba} \to \mathtt{ab}, \epsilon \to \mathtt{c}$) contains *two* operations: first, it swaps a and b, and then it inserts c between them. Note that the insertion changes the already modified string. However, if subsequent modifications are ruled out, a shortest edit script that transforms "ba" into "acb" (e.g., $\mathtt{b} \to \epsilon, \epsilon \to \mathtt{c}, \epsilon \to \mathtt{b}$) contains *three* edit operations. Thus, the unrestricted edit distance is equal to two, while the restricted distance is equal to three.

The Damerau-Levenshtein distance from "ab" to "ba" as well as from "ab" and "acb" is equal to *one*. Therefore, restricted Damerau-Levenshtein distance does not satisfy the triangle inequality because

$$2 = \mathrm{ED}(\mathtt{ab}, \mathtt{ba}) + \mathrm{ED}(\mathtt{ab}, \mathtt{acb}) < \mathrm{ED}(\mathtt{ba}, \mathtt{acb}) = 3$$

■

*2.2.4. Optimal Alignment.* The problem of efficient computation of the unrestricted Damerau-Levenshtein function was solved by Lowrance and Wagner [1975]. In what follows, we give formal definitions and survey the classic dynamic programming algorithm for the computation of the Levenshtein distance and its extension suggested by Lowrance and Wagner [1975].

Let strings $p$ and $s$ be partitioned into the same number of possibly empty substrings: $p = p_1 p_2 \dots p_l$ and $s = s_1 s_2 \dots s_l$, such that $p_t \to s_t \in \mathbb{B}$. Additionally we assume that $p_t$ and $s_t$ cannot be empty at the same time. We say that this partition defines an *alignment* $A = (p_1 p_2 \dots p_l, s_1 s_2 \dots s_l)$ between $p$ and $s$, in which substring $p_t$ is aligned with substring $s_t$.

The alignment represents the *restricted* edit script $E = p_1 \to s_1, p_2 \to s_2, \dots, p_l \to s_l$. We define the *cost of alignment* $A$ as the cost of corresponding edit script and denote it as $\delta(A)$:

$$\delta(A) = \sum_{t=1}^{l} \delta(p_t \to s_t) \qquad (2)$$

An *optimal alignment* is an alignment with the minimum cost.

Fig. 1: An alignment of strings “wadr” and “sword”

$$\begin{array}{cccc} \epsilon & \texttt{w} & \texttt{a} & \texttt{dr} \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ \texttt{s} & \texttt{w} & \texttt{o} & \texttt{rd} \end{array}$$

An example of an *optimal* alignment between strings “wadr” and “sword” is presented in Figure 1. The corresponding edit script consists of insertion $\epsilon \to$ s, substitution a $\to$ o, and transposition dr $\to$ rd.

It can be also seen that there exists a one-to-one mapping between a set of restricted edit scripts and a set of alignments: each restricted edit script with the minimum cost represents an alignment with the minimum cost and vice versa. Thus, we can substitute the task of finding optimal restricted edit distance with the task of finding the cost of the optimal alignment.

*2.2.5. Edit Distance Computation.* The dynamic programming algorithm to compute the cost of an optimal alignment was independently discovered by several researchers in various contexts, including speech recognition [Vintsyuk 1968; Velichko and Zagoruyko 1970; Sakoe and Chiba 1971] and computational biology [Needleman and Wunsch 1970] (see Sankoff [2000] for a historical perspective). Despite the early discovery, the algorithm was generally unknown before a publication by Wagner and Fischer [1974] in a computer science journal.

The main principle of the algorithm is to express the cost of alignment between strings $p$ and $s$ using costs of alignments between their prefixes. Consider the prefix $p_{[1:i]}$ of the length $i$ and the prefix $s_{[1:j]}$ of the length $j$ of strings $p$ and $s$, respectively. Assume that $A = (p_1 p_2 \ldots p_l, s_1 s_2 \ldots s_l)$ is an *optimal* alignment between $p_{[1:i]}$ and $s_{[1:j]}$, whose cost is denoted by $C_{i,j}$.

Using Equation (2) and the definition of optimal alignment it is easy to show that $C_{i,j}$ can be computed using the following generic recursion [Ukkonen 1985a; Veronis 1988]:[8]

$$\begin{aligned} &C_{0,0} = 0 \\ &C_{i,j} = \min\{\delta(p_{[i':i]} \to s_{[j':j]}) + C_{i'-1,j'-1} \,|\, p_{[i':i]} \to s_{[j':j]} \in \mathbb{B}\} \end{aligned} \tag{3}$$

Recursion (3) is an example of a *dynamic programming* solution. The set of $(|p|+1) \cdot (|s|+1)$ numbers $\{C_{i,j}\}$ is commonly referred to as a *dynamic programming matrix* (or shortly *DP matrix*). It can be also seen that:

— The cost of alignment between strings $p$ and $s$ is equal to $C_{|p|,|s|}$;
— All optimal alignments can be recovered by backtracking through Recursion (3).

Let us now consider the case of Levenshtein distance, where $p_{[i':i]} \to s_{[j':j]}$ is a unit-cost single character insertion, deletion, or substitution. Therefore,

$$\delta(p_{[i':i]} \to s_{[j':j]}) = [p_{[i':i]} \neq s_{[j':j]}],$$

where $[X]$ is equal to one, if condition $X$ is true and is zero otherwise. Furthermore, there are three possible combinations of $i'$ and $j'$, which correspond to deletion, insertion, and substitution, respectively:

— $i' = i - 1$ and $j' = j$.
— $i' = i$ and $j' = j - 1$;
— $i' = i - 1$ and $j' = j - 1$.

[8]We also suppose that this simple generalization was independently discovered by many other researchers.

Considering these simplifications, we can rewrite generic Recursion (3) for Levenshtein distance as follows:

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + \left[p_{[i]} \neq s_{[j]}\right], & \text{if } i, j > 0 \end{cases} \tag{4}$$

It can be seen that the DP matrix can be computed by a column-wise top-down traversal (one column at a time) or by row-wise left-to-right traversal (one row at a time) in $O(|p| \cdot |s|)$ time and space. The cost of optimal alignment alone can be computed in $O(|p| \cdot |s|)$ time and $O(\min(|p|, |s|))$ space (the algorithm has to remember only last column or row to compute a new one) [Navarro 2001a].

According to Observation 2.10, the unrestricted Levenshtein distance is equal to restricted Levenshtein distance. The restricted edit distance, on the other hand, is equal to the cost of an optimal alignment. Therefore, Recursion (4) calculates the *unrestricted* Levenshtein distance. A common misconception is that the following straightforward generalization of Recursion (4) correctly computes the *unrestricted* Damerau-Levenshtein distance in all cases:

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + \left[p_{[i]} \neq s_{[j]}\right], & \text{if } i, j > 0 \\ C_{i-2,j-2} + 1, & \text{if } p_{[i]} = s_{[j-1]}, p_{[i-1]} = s_{[j]} \text{ and } i, j > 1 \end{cases} \tag{5}$$

However, Recursion (5) evaluates only the *restricted* edit distance, which is not always equivalent to the unrestricted edit distance. For instance, the distance between strings "ba" and "acb" computed using Recursion (5) is equal to three, while the unrestricted Damerau-Levenshtein distance between these strings is equal to two.

As shown by Lowrance and Wagner [1975], the unrestricted Damerau-Levenshtein distance can be calculated as the cost of the restricted edit distance (i.e., the cost of an optimal alignment), where $\mathbb{B}$ consists of unit-cost single-character insertions, deletions, substitutions, and operations $aub \to bva$ at the cost $|u| + |v| + 1$.[9] Given this definition, generic Recursion (3) can be rewritten for Damerau-Levenshtein distance as follows [Lowrance and Wagner 1975]:

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + [p_i \neq s_j], & \text{if } i, j > 0 \\ \min\limits_{\substack{0 < i' < i,\, 0 < j' < j \\ p_{[i]} = s_{[j']},\, p_{[i']} = s_{[j]}}} C_{i'-1,j'-1} + (i - i') + (j - j') - 1 \end{cases} \tag{6}$$

In addition, Lowrance and Wagner demonstrated that the inner minimum in Recursion (6) is achieved at the largest $i' < i$ and $j' < j$ that satisfy $p_{[i]} = s_{[j']}$ and $p_{[i']} = s_{[j]}$. Lowrance and Wagner proposed an algorithm to compute Recursion (6) in $O(|p| \cdot |s|)$ time. Their solution takes advantage of the fact that the DP matrix is computed row-wise top-down (or column-wise left-to-right), which allows us to find $i'$ and

[9]Note that we can restrict the set of operations $\{aub \to bva\}$ to the subset $\{aub \to bva \,|\, u = \epsilon \text{ or } v = \epsilon\}$. If both $|u| \geq 1$ and $|v| \geq 1$, $aub$ can be transformed into $bva$ using $\max(|u|, |v|) + 2 \leq |u| + |v| + 1$ insertions, deletions, and substitutions. Thus, there is an alternative optimal edit script that does no contain the operation $aub \to bva$.

$j'$ that satisfy $p_{[i]} = s_{[j']}$ and $p_{[i']} = s_{[j]}$ iteratively, using only additional $|\Sigma|+1$ counters. See the pseudo-code in Appendix D.3 or the article by Lowrance of Wagner [1975] for details.

### 2.3. Frequency Distance

Consider a dataset comprised of $m$-dimensional vectors. Given query vector $z$, the problem of approximate vector space searching consists in finding elements similar to $z$. Depending on the definition of similarity, the problem has several variations, among which satisfying *partial match queries* and *region queries* (also known as *orthogonal range queries*) received most attention. The problem of satisfying *frequency-distance* queries is less known, but it provides the basis for several filtering string search methods.

The frequency distance originates from the concept of *counting filter* introduced by Grossi and Luccio [1989] for the purpose of substring matching. This concept was improved by several other researchers including Jokinen et al. [1996] and Navarro [1997a].

Grossi and Luccio proved the following: if the pattern $p$ matches a substring $t_{[i:j]}$ with at most $k$ errors, then the substring $t_{[j-|p|+1:j]}$ includes at least $|p|-k$ characters of $p$ (with multiple occurrences accounted for).

In the case of complete word matching, the counting filter can be used reciprocally: if $\mathrm{ED}(p, s) \le k$, then the string $p$ should contain at least $|s|-k$ characters from the string $s$ and the string $s$ should contain at least $|p|-k$ characters from the string $|p|$. The reciprocal counting filter was described by Kahveci and Singh [2001]. In what follows, we give a formal definition of the frequency distance and explain its relation to the edit distance.

#### 2.3.1. Definition of Frequency Distance.

The *positive frequency distance* $\mathrm{FD}_+(x, y)$ and negative frequency distance $\mathrm{FD}_-(x, y)$ between $x$ and $y$ are defined by the following formulae:

$$\mathrm{FD}_+(x, y) = \sum_{x_i > y_i} x_i - y_i = \sum_{i=1}^{m} [x_i > y_i] \cdot (x_i - y_i) \tag{7}$$

$$\mathrm{FD}_-(x, y) = \sum_{x_i < y_i} y_i - x_i = \sum_{i=1}^{m} [x_i < y_i] \cdot (y_i - x_i), \tag{8}$$

where $[X]$ is equal to one if condition $X$ is true and is zero otherwise. $\mathrm{FD}(x, y) = \max(\mathrm{FD}_+(x, y), \mathrm{FD}_-(x, y))$ is *frequency distance*.[10]

#### 2.3.2. Relationship to Edit Distance.

Given strings $p$ and $s$, the frequency distance between the corresponding *unigram* frequency vectors $\mathrm{vect}(p)$ and $\mathrm{vect}(s)$ is a *lower bound* for both the restricted and unrestricted edit distance between $p$ and $s$ [Kahveci and Singh 2001]:

$$\mathrm{FD}(\mathrm{vect}(p), \mathrm{vect}(s)) \le \mathrm{ED}\,(p, s) \tag{9}$$

Inequality (9) indicates that the problem of approximate dictionary search can be reduced to searching in a vector space. In this approach a string is represented by its unigram frequency vector. Because frequency distance between unigram frequency

[10]It is indeed a distance, which follows from the equation $\mathrm{FD}(x, y) = (|d_M(x, \vec{0}) - d_M(y, \vec{0})| + d_M(x, y))/2$, where $d_M(x, y) = \sum_i |x_i - y_i|$ is Manhattan distance, and $\vec{0}$ represents the null vector all of whose elements are zero.

vectors is a lower bound for the edit distance between respective strings, we can process *unigram frequency distance queries* in the vector space with the same threshold value $k$. Given the pattern $p$ and the maximum allowed edit distance $k$, the search procedure finds all strings such that:

$$\text{FD}(\text{vect}(p), \text{vect}(s)) \le k.$$

Note that the projection of strings to the corresponding frequency vectors does not preserve edit distance: for instance, $\text{ED}(\texttt{abc}, \texttt{bca}) = 2$, whereas $\text{FD}(\text{vect}(\texttt{abc}), \text{vect}(\texttt{bca})) = 0$. Therefore, a search in the vector space may not satisfy a query accurately; rather, it would provide a *candidate list*, which needs to be verified element-wise using the edit distance.

Mapping strings to frequency vectors is an example of a *filtering method* that consists in using a less computationally expensive algorithm to filter out the majority of data elements.

In this survey, we use frequency vectors to demonstrate the principles of searching in a vector space using the frequency distance. Because

$$\text{FD}(\text{signat}(p), \text{signat}(s)) \le \text{FD}(\text{vect}(p), \text{vect}(s)),$$

the presented properties are also valid in the signature-based vector space searching. In particular, one can see that the frequency distance between string signatures is also a lower bound for the edit distance between respective strings.

*2.3.3. Threshold for Searching in Vector Spaces.* A frequency distance between unigram frequency vectors is always a lower bound for the edit distance between respective strings, see Inequality (9). This property, however, does not hold for $q$-gram frequency vectors if $q > 1$. Therefore, if we transform strings to $q$-gram frequency vectors, we have to choose a larger threshold for searching in the vector space. In what follows, we calculate this threshold.

Each string $s$ can be represented by a sequence of $q$-grams $s_{[1:q]}, s_{[2:q+1]}, \ldots, s_{[n-q+1:n]}$ contained in it. This sequence is itself a string over the alphabet $\Sigma^q$, which we have denoted by $q$-grams$(s)$. If $q = 1$, then $q$-grams$(s) = s$.

Consider strings $p$ and $s$ and their respective $q$-gram sequences: $q$-grams$(p)$ and $q$-grams$(s)$. Let these $q$-gram sequences be converted to frequency vectors vect($q$-grams$(p)$) and vect($q$-grams$(s)$). Since $q$-gram sequences are strings over alphabet $\Sigma^q$, from Inequality (9) it follows that the frequency distance between two $q$-gram frequency vectors lower bounds the edit distance between respective $q$-gram sequences:

$$\text{FD}(\text{vect}(q\text{-grams}(p)), \text{vect}(q\text{-grams}(s))) \le \text{ED}(q\text{-grams}(p), q\text{-grams}(s)) \qquad (10)$$

For $q > 1$, a basic edit operation may change more than one $q$-gram. According to Jokinen and Ukkonen [1991], the number of $q$-grams modified by a single insertion, deletion, or substitution is at most $q$. It can be also seen that a single transposition can modify up to $(q + 1)$ $q$-grams.[11]

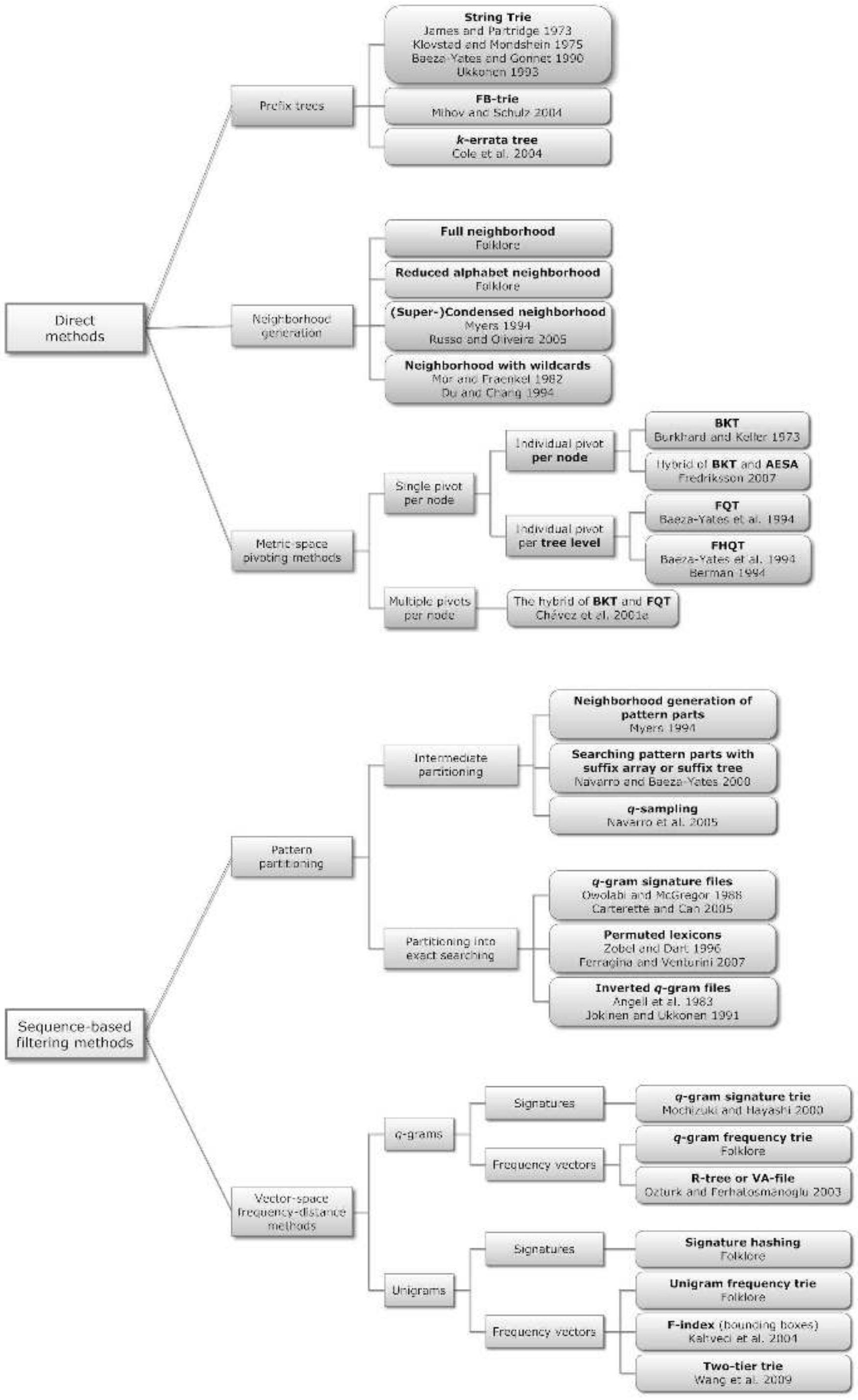
Combining these observations and Inequality (10), we obtain the following maximum allowed frequency distance for searching in the vector space:

$$k \times (q + [\text{transposition-aware and } q > 1]), \qquad (11)$$

where $k$ is the maximum allowed edit distance.

[11]Consider, for example, string “abcd” and transposition $\texttt{bc} \rightarrow \texttt{cb}$, which modifies all three bigrams of the string.

Fig. 2: Taxonomy of search methods



## 3. TAXONOMY OUTLINE

In order to improve understanding of the field, we introduce a taxonomy of approximate search methods. It is an extension of the taxonomy of Navarro et al. [2001], who suggest to classify approximate search methods into neighborhood generation, partitioning into exact searching, and intermediate partitioning methods. In this survey, all indexing methods are divided into two main groups:

— Direct methods;
— Sequence-based filtering methods.

Each category can be further subdivided based on characteristics specific to the method type.

*Direct search methods* look for complete patterns. They include: *prefix trees*, *neighborhood generation*, and *metric space* methods. A neighborhood generation method constructs a list of words within a certain edit distance from a search pattern. Then the words from this list are searched for exactly. Both prefix trees and most metric space methods recursively decompose a dictionary into mutually disjoint parts, i.e., they create a hierarchy of space partitions. A search algorithm is a recursive traversal of the hierarchy of partitions. At each recursion step, the search algorithm determines whether it should recursively enter a partition or backtrack, i.e., abandon the partition and return to a point where it can continue searching. Such a decision is made on the basis of proximity of the search pattern and strings inside the partition.

A *filtering search method* has a filtering step and a checking step. During the filtering step, the method builds a set of candidate strings. The checking step consists in element-wise comparison of the search pattern and candidate strings. Most filtering methods compute the edit distance between a candidate string $s$ and the search pattern $p$ to discard strings $s$ such that $\mathrm{ED}(p, s) > k$. Alternatively, it is possible to use a fast on-line algorithm, which essentially applies a second filter.

Filtering algorithms can be characterized by a *filtering efficiency*. It is informally defined as a fraction of dictionary strings that are discarded at the filtering step. A complementary measure is filtering *inefficiency*, which is defined as one minus filtering efficiency. Filtering inefficiency is computed as the average number of verifications during the checking step divided by the number of dictionary strings.

*Sequence-based* filtering methods employ short string fragments to discard nonmatching strings without evaluating the edit distance to a search pattern directly. We call these fragments *features*. Sequence-based filtering methods can be divided into *pattern partitioning* and *vector-space* methods. Pattern partitioning methods rely on direct indexing of and searching for string fragments, e.g., using an inverted file. Vector-space methods involve conversion of features (usually $q$-grams and unigrams) into frequency vectors or signatures. Then frequency vectors and signatures are indexed using general vector space methods.

In addition, there are *hybrid methods* that integrate several access methods to improve retrieval time. Many hybrid methods also rely on filtering. Hybrid methods are very diverse: we do not discuss them separately from other algorithms.

The taxonomy of search methods is presented in Figure 2.[12] In the following sections, we discuss search methods in detail, starting with a brief description of exact search and sequential search algorithms that are used as a part of indexing search methods.

[12]Note that in some cases, provided references describe sequence-oriented search methods or search methods that do not use the edit distance. However, the presented ideas can be also applied to dictionary search methods with the edit distance.

## 4. EXACT SEARCHING

There are two mainstream approaches to exact string searching: *hashing* and *tries*. Tries allow one to find the pattern $p$ in $O(|p|)$ time in the worst case. Unlike tries, most hashing methods have retrieval time proportional to the pattern length only on average.

Tries may or may not be faster than hashing methods depending on a dataset, environment and implementation specifics. For instance, our implementation of a *full trie* slightly outperforms a hashing implementation supplied with a GNU C++ compiler for all natural language datasets used in our tests. However, hashing is about twice as fast as a full trie implementation for DNA data. It is also about 2-3 times faster than our *path-compressed* implementation of a trie (see Section 4.2 below for a description of trie variants).

The advantage of tries is that they efficiently support a prefix query, i.e., a request to find all strings that have a specified prefix. A related search problem consists in finding all occurrences of a substring $p$ in a string $s$. This problem can be solved with the help of *suffix trees* or *suffix arrays*. In the following sections, we briefly discuss hashing, tries, suffix trees, and suffix arrays.

### 4.1. Hashing

Hashing is a well-known method extensively covered in the literature, see, e.g., [Knuth 1997] for a detailed discussion. An underlying idea is to map strings to integer numbers from 1 to some positive integer $M$ with the help of a *hash function*. Given a storage table of size $M$ (known as a *hash table*) and a hash function $h(s)$, the string $s$ is stored in the cell number $h(s)$. Associated data can be stored together with the string.

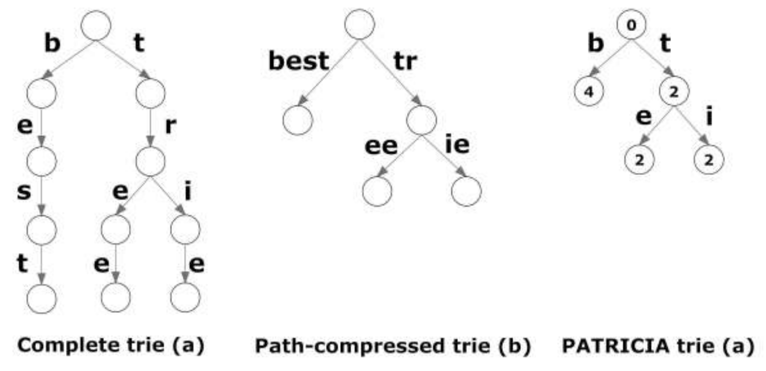
A situation when two (or more strings) are mapped to the same cell is called a *collision*. There are two main methods to resolve collisions. The first method is *chaining*. It involves storing collided strings in the form of a list. The second method consists in finding a free cell other than $h(s)$ using a predetermined rule, also known as a probe sequence. This method is called open addressing. If no free cell can be found, the hash table is resized and its elements are reallocated.

These classic methods are fast only on average. Perfect hashing eliminates collisions by constructing a hash function that maps no pair of dictionary strings to the same number [Fredman et al. 1984]. Perfect hashing retrieves the pattern $p$ in $O(|p|)$ time in the worst case. It works best for static dictionaries, but there are also dynamic modifications [Dietzfelbinger et al. 1994]. An alternative method that finds the pattern in $O(|p|)$ worst-case time is cuckoo hashing [Pagh and Rodler 2001].

A straightforward implementation of hashing based on open addressing is not space-economical. To efficiently resolve collisions, the table should be big enough so that a significant fraction, e.g., one half, of all cells are empty. It is possible to trade space for performance through resolving collisions via chaining.

In the case of static dictionaries, one can employ a *minimal perfect hashing* [Botelho et al. 2007]. A minimal perfect hash function $h(s)$ maps a set of $M$ strings to the numbers from 1 to $M$ without collisions. Assume that dictionary strings $s_i$ are stored in increasing order of the values of the minimal perfect hash function $h(s_i)$. Then the hash table contains a sequence of increasing string offsets, which can be efficiently compressed. The most efficient compression can be achieved with the help of a prefix-sum data structure [Elias 1974; Raman et al. 2007]. It requires at most $N(\log_2 \lambda + C) + o(N)$ bits of storage, while allowing one to retrieve the $i$-th string offset in constant time ($\lambda$ is the average length of a dictionary string and $C \leq 2$ is a small constant).

Fig. 3: Three modifications of tries storing keys: "`best`", "`tree`", and "`trie`"



### 4.2. Trie

A *trie* (or a *prefix tree*) is a tree where strings with common prefixes are grouped into subtrees. Edges of the prefix tree have string *labels*. Each node $\xi$ is associated with the string $s$, formed by concatenating labels on the way from the root to $\xi$. It is common to say that $\xi$ *spells* the string $s$. The *level* of the node $\xi$ is the length of the string $s$. Tries can answer prefix queries in a straightforward way.

For simplicity of exposition, we assume that the dictionary $W$ is prefix-free: no string in $W$ is a proper prefix of another string. This condition can be ensured by extending an alphabet with a special character \$ and appending it to every string in the set. In a prefix-free dictionary, only a leaf spells a complete dictionary string, while an internal node spells a *proper* prefix of a dictionary string.

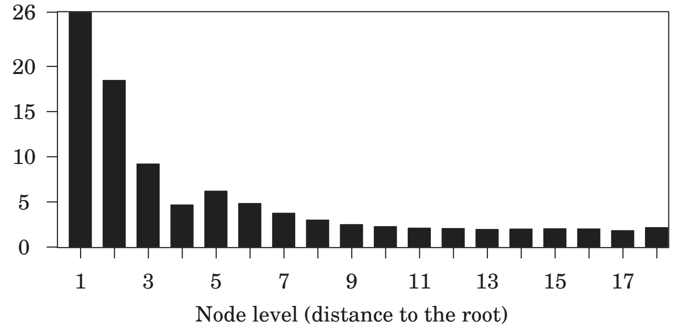
Tries have three main modifications, which are presented in Figure 3. In a *complete trie* (full trie) every label is exactly one character long and all characters are stored in the tree, without omissions (see Panel (a) in Figure 3). A representation of a complete trie has many pointers to child nodes, which is not space-efficient. A number of compression methods have been proposed in the literature.

*Path compression* is an early lossless compression algorithm that consists in merging of single-child nodes with their children. Consider a sequence of nodes $\xi_1, \xi_2, \ldots, \xi_l$, i.e., a *path*, with edges labeled with strings $s_1, s_2, \ldots, s_l$ such that $\xi_{i+1}$ is the only child of $\xi_i$ for each $i < l$. This path can be compressed by replacing nodes $\{\xi_i\}$ and respective edges with a single node. This node has a single outgoing edge labeled by the string $s_1 s_2 \ldots s_l$. It can be seen that a path-compressed trie has fewer pointers than the corresponding complete trie. See Panel (b) in Figure 3 for an example.

*Level compression* is a more recent method [Nilsson and Karlsson 1999]. Consider a complete trie, where a node at level $l$ and all of its descendants at levels $l+1, l+2, \ldots, l+m-1$ have exactly $|\Sigma|$ children, i.e., one child per alphabet character. Then we can replace these nodes with a single node of degree $|\Sigma|^m$. Level compression works well for binary tries and/or DNA , however, it is not very useful in the case of large alphabets and natural languages, where tries are usually sparse (see Figure 4).

A PATRICIA trie [Morrison 1968; Gwehenberger 1968] is an example of a lossy compression method, where each label $s$ of a path-compressed trie is replaced with its first character $s_{[1]}$ and label length $|s|$, also known as a *skip value*. See Panel (c) in Figure 3 for an example (skip values are depicted in circles). Because the PATRICIA trie is based on lossy compression, the search algorithm should refer to the original data to avoid false positives (only one verification is needed for each string found).

Fig. 4: The average number of children per trie level (Synthetic English data, 3.2M strings)



A number of compression methods rely on storing trie nodes in a pre-specified traversal order. Consider, for example, a recursive traversal procedure that visits trie nodes level-wise in increasing order of levels. The nodes of the same level are visited in increasing alphabetic order of strings they spell.

Assume that nodes of the trie are stored in memory in order of their traversal as records of the same size. Let, in addition, the record of node $\xi$ (of level $l$) contain the following data:

— The bitmask of size $|\Sigma|$, where the $i$-th bit is equal to one if $\xi$ has an outgoing edge labeled with $\Sigma_i$;
— The total number of children of all nodes at level $l$ that are stored prior to $\xi$.

If, in addition, there is an array of pointers to the first node of each level, this data unambiguously defines locations of all trie nodes.

The data structure described is known as C-trie [Maly 1976]. C-trie achieves a compact representation by essentially sharing pointers among children. It uses $N(|\Sigma| + \log_2 N)$ *bits* of space, which is often less than the size of the dictionary.[13] Benoit et al. [2005] proposed a more compact data structure that employs storing trie nodes in a traversal order. It uses only $N(\lceil \log_2 |\Sigma| \rceil + 2) + o(N) + O(\log \log |\Sigma|)$ *bits*, which is close to the information theoretic lower bound. Such compact trees are often called *succinct*.

A disadvantage of all methods that rely on a traversal-order trie representation is that updates can be expensive. Consider, for example, a C-trie such that no string contains character "a" and $m$ is the maximum length of the stored strings. Consider also the string $s =$ "aa...a" of the length $m$: the string $s$ does not belong to the trie. It can be seen that an insertion of $s$ requires an update of every C-trie node.

Another compression method is based on the observation that many subtrees in tries are isomorphic. Therefore, we can keep only one instance of isomorphic subtrees and replace the other instances with a pointer to the unique representative. As a result, the dictionary can be represented by a compact finite state machine, which can be computed efficiently and incrementally [Daciuk et al. 2000]. A drawback of this approach is that a single final state might correspond to several dictionary strings. Therefore, the data structure is not associative and cannot store auxiliary information such as string identifiers.

[13]For instance, consider a Russian dictionary containing less than four million entries. Because $|\Sigma| = 33$ and $\log_2 N < 22$, storage requirement of C-trie is approximately 55 bits (or seven bytes) per string, while the average length of the dictionary string is about 10 characters.

### 4.3. Suffix Tree and Suffix Array

Because the trie supports prefix queries, it is possible to efficiently answer substring queries on a string $s$ by storing all suffixes of $s$ in the trie. Typically the PATRICIA trie is used. The resulting data structure is called a *suffix tree*.

The suffix tree can be constructed in linear space and time with respect to the string length [Weiner 1973; McCreight 1976; Ukkonen 1995]. A classic, i.e., not compressed, implementation of a suffix tree requires as much as 8-10 bytes per character on average [Kurtz 1999; Giegerich et al. 2003].

A more space-economical variant of a suffix tree is a *suffix array*, which is essentially a sorted array of string suffixes [Manber and Myers 1990]. More specifically, a suffix array of the string $s$ is a sorted array of starting positions $i$ of suffixes $s_{[i:|s|]}$. This array is sorted the lexicographic order with respect to $s_{[i:|s|]}$. Assuming that an integer requires 4 bytes of the storage, a suffix array uses at least 4 bytes per character, which is still a big overhead.[14]

There exist compressed versions of suffix arrays [Grossi and Vitter 2005] and suffix trees [Sadakane 2007; Russo et al. 2008; Fischer et al. 2008; Russo et al. 2009]. The proposed methods have a wide range of space-time tradeoffs: the most compact indices require space comparable to the size of the compressed source data, however, they are also the slowest ones (see [Cánovas and Navarro 2010] for experimental comparisons).

Suffix trees and suffix arrays can be used to answer substring dictionary queries in a straightforward way. The solution consists in appending a special separator character \$ to every dictionary string, concatenating all dictionary strings, and subsequent indexing using a suffix tree or array (see, e.g., [Bieganski et al. 1994]).

A *permuted lexicon* is another method that supports substring searching [Zobel and Dart 1995]. In addition, this method allows one to efficiently answer *prefix-suffix* queries, i.e., requests to find all dictionary strings in the form $u*v$, where the wildcard "$*$" represents any (possibly empty) string.

The indexing algorithm of the permuted lexicon involves appending the separator to every dictionary string and indexing all its cyclic rotations. For example, a string "`beta`" would be represented by "`beta$`", "`eta$b`", "`ta$be`", "`a$bet`", and "`$beta`".

A naive implementation of this method that actually stores all cyclical string rotations, e.g., using hashing, is space-inefficient. A more compact data structure is essentially a suffix array: a pointer to the character $s_{[i]}$ in the text representation of the dictionary corresponds to the string $s$ rotated by $i$ characters. Note that the pointers are to be sorted by their respective rotations. This is different from the suffix array built over concatenated dictionary strings, where pointers are sorted by the suffixes of the string produced via concatenation.

A compressed form of the permuted lexicon was recently proposed by Ferragina and Venturini [2007]. Their method employs the Burrows-Wheeler transform, which sorts cyclical rotations of the string obtained by concatenating \$-separated dictionary strings.

## 5. SEQUENTIAL SEARCHING

The checking step of several filtering methods involves verification of whether the edit distance between a candidate string $s$ and the search pattern $p$ does not exceed $k$. A straightforward implementation of the checking step would involve computation of the dynamic programming matrix (see Section 2.2.5). The dynamic programming approach is unexcelled in flexibility. However, in the domain of approximate string searching for

[14]Manber and Myers [1990] also describe implementations that require more space.

short patterns and limited number of errors, this flexibility is largely unclaimed, not to mention its high computational costs.

Because for most surveyed methods it is not necessary to calculate an accurate value of the edit distance, the dynamic programming algorithm can be modified to compute only $2k + 1$ main diagonals of the dynamic programming matrix [Ukkonen 1985b]. Several faster verification methods rely on *bit-parallelism* [Wu and Manber 1992b; 1992a; Navarro and Raffinot 2000; Navarro 2001b]. In addition, bit-parallelism can be used to efficiently compute the dynamic programming matrix [Myers 1999; Hyyrö 2005]. The key idea of bit-parallel methods is to simultaneously update several small numbers packed into a single computer word using a few processor instructions. It can be seen that the degree of parallelization increases with the size of the computer word.

The *Shift-AND* is an early bit-parallel algorithm proposed by Dömölki [1968] for the purpose of exact string matching. Later, a variant of this algorithm, known as the *Shift-OR*, was rediscovered by Baeza-Yates and Gonnet [1992]. They also extended the Shift-AND to the case of the Hamming distance, multiple-pattern searching, and searching for simple regular expressions.

Wu and Manber [1992a] proposed a generalization of the Shift-AND for searching with the edit distance. Their method is based on simulation of a non-deterministic *Levenshtein automaton*. Given the pattern $p$ and the maximum allowed distance $k$, the Levenshtein automaton is a finite state machine that accepts only strings $s$ such that $\mathrm{ED}(p, s) \leq k$. In the Shift-AND algorithm, the states of the non-deterministic automaton are represented by the bits of computer words: unit bit values correspond to active states and zero bit values correspond to inactive states.

Another approach to efficient approximate verification involves a *deterministic* Levenshtein automaton. Kurtz [1996] and Navarro [1997b] suggested to evaluate the automaton in a lazy way. Mihov and Schulz [2004] proposed to use a *universal* deterministic Levenshtein automaton, which is fully constructed before searching. This is a modification of Wu et al.'s [1996] automaton for the purpose of complete string matching. The automaton is universal in the sense that it does not depend on $p$ and $s$.

For a detailed discussion of sequential search algorithms we refer the reader to the surveys by Jokinen et al. [1996], Navarro [2001a], and Hyyrö [2003a; 2003b].

In our experiments, we use a combination of algorithms implemented in agrep [Wu and Manber 1992b; 1992a], which we denote by magrep1 and magrep2. They are known to be fast for short patterns and small number of errors. In Section 8.2.3 we verify the efficiency of these algorithms on our data.

Both magrep1 and magrep2 use a two-step on-line filtering procedure, where the filtering consists in splitting the pattern and searching for pattern parts exactly. The methods use two algorithms for exact multi-pattern searching: magrep1 uses a variant of the Shift-OR algorithm as described by Wu and Manber [1992b]; magrep2 uses a Boyer-Moore like algorithm proposed by Wu and Manber in another paper [1992a].

The checking step of both magrep1 and magrep2 employs a transposition-aware variant of the Shift-AND algorithm [Hyyrö 2003a], which also simulates the non-deterministic Levenshtein automaton. The underlying automaton is modified for the task of complete string matching: it does not not contain a self-loop in the start state and keeps only states that correspond to $2k + 1$ main diagonals of the dynamic programming matrix.

## 6. DIRECT INDEXING METHODS

### 6.1. Prefix Trees

*6.1.1. String Trie.* An important property of a prefix tree is that every approximate occurrence of the search pattern $p$ can be found by a recursive traversal of the tree

**Algorithm 1** Searching in the Trie

```
(1) If $\xi_{curr}$ is a leaf, the algorithm backtracks after checking whether $T_{curr}$ is an accept
    state and outputting the string $s_{curr}$ whenever this check succeeds;
(2) If $\xi_{curr}$ is an internal node, the Step 3 is repeated for every child node $\xi_{child}$ of the
    node $\xi_{curr}$;
(3) Let $v$ be the label of the edge that leads from $\xi_{curr}$ to $\xi_{child}$. The Levenshtein au-
    tomaton consumes the characters of the label $v$ and moves to the state $T'$. If $T'$ is
    the fail state, the algorithm backtracks. Otherwise, it proceeds recursively to Step
    1 with $\xi_{curr} = \xi_{child}$, $s_{curr} = s_{curr}v$, and $T_{curr} = T'$.
```

starting from the root. An early implementation of this idea was described by James and Partridge [1973], who proposed an algorithm that is capable of recovering from a mismatch by recursively considering substitutions, insertions, and deletions along the search path. This algorithm does not rely on the edit distance and uses domain-specific heuristics to produce the most likely match (in particular, the probability of encountering a string). Klovstad and Mondshein [1975] described a trie-based search algorithm that obtains a similarity score for any match found. Though few details are given, the score is apparently computed as the extended edit distance (see Section 2.2) between dictionary and pattern strings, normalized by lengths of pattern strings.

To the best of our knowledge, the trie-based search algorithm presented in this section originated from computational biology and was rediscovered many times. Some of the early descriptions were given by Baeza-Yates and Gonnet [1990][15] and by Ukkonen [1993]. This method is best described as a recursive parallel traversal of the trie and the deterministic Levenshtein automaton. Let $s$ be a string that is a *not* a prefix of an approximate match of $p$ with at most $k$ errors. If the Levenshtein automaton "consumes" $s$, it reaches the special fail state.

The trie-based search algorithm is as a recursive procedure that keeps a node $\xi_{curr}$, a string $s_{curr}$, and a state $T_{curr}$ of the Levenshtein automaton. The recursion starts with $\xi_{curr}$ pointing to the root node, $s_{curr} = \epsilon$, and $T_{curr} = T_0$ ($T_0$ is the start automaton state). It unfolds as shown in Algorithm 1.

This algorithm does not necessarily require the deterministic Levenshtein automaton and can use any verification method. In particular, our implementation of the string trie is based on the bit-parallel algorithm Shift-AND [Wu and Manber 1992a] (see Section 5).

Another approach to verifying whether the string $s$ starts an approximate match for the pattern $p$ consists in incremental column-wise computation of the dynamic programming matrix (DP matrix) using Recursion (4), p. 11.

In the case of the Levenshtein distance, this algorithm checks whether the last column contains *essential elements*, i.e., elements with values less than or equal to $k$. It is equivalent to the condition that a prefix of $p$ matches $s$ within $k$ errors. If all column elements are larger than $k$, the string $s$ cannot start an approximate occurrence and the algorithm backtracks. Whenever the search procedure reaches a leaf, it checks whether the bottom element of the last column of the DP matrix is less than or equal to $k$.

In the case of the Damerau-Levenshtein distance and transposition-aware searching, the search procedure computes the DP matrix using Recursion (5). Since values of elements in the current column depend on values of elements in both the last and

[15]A conference version of this paper was published nine years later [Baeza-Yates and Gonnet 1999].

Fig. 5: DP matrices computed during searching in the trie

| | b | e | s | t |
|---|---|---|---|---|
| e | 1 | 1 | 2 | 3 |
| s | 2 | 2 | 1 | 2 |
| t | 3 | 3 | 2 | 1 |

(a) For strings "`est`" and "`best`"

| | t | r |
|---|---|---|
| e | 1 | 2 |
| s | 2 | 2 |
| t | 2 | 3 |

(b) For strings "`est`" and "`tr`"

the next to last column, the search procedure checks whether at least one of two last columns contains an essential element.

Consider, for instance the trie on Panel (b) of Figure 3, p. 17. Assume that we search for pattern "`est`" within one error ($k = 1$) using the transposition-*unaware* method. The search procedure starts from the root and proceeds to the child on the left, which spells string "`best`". At this point, the search procedure calculates the DP matrix column by column. The transposition-unaware search procedure keeps only the last column of the DP matrix, but for clarity of exposition we present the complete DP matrix (see Panel (a) of Figure 5). Also note that we omit the row that corresponds to $i = 0$ and the column that corresponds to $j = 0$ in Recursion (5).

Each column of the matrix on Panel (a) of Figure 5 has one essential value. Therefore, the search procedure cannot leave the branch until it reaches the terminal node. The bottom element of the last row of the DP matrix is equal to 1. Because the value of this element does not exceed the threshold of $k = 1$, the search procedure reports an approximate match: "`best`".

Next, the search procedure returns to the root and descends to the child node on the right. This node spells string "`tr`". Then it computes the part of the DP matrix that corresponds to string "`tr`" (see Panel (b) of Figure 5). The first column contains an essential element. Therefore, the search procedure has to compute the second column. The second column does not contain essential elements, which results in the search procedure abandoning the branch and completing execution.

*6.1.2. FB-trie.* In this section, we review another approach to improving the retrieval time that combines pattern partitioning and tries. It requires a pair of tries: the trie built over the dictionary strings and the trie built over the reversed dictionary strings. The idea of using an index built over the reversed dictionary strings was proposed by Knuth [1973] (see also [Knuth 1997], p. 394).

There exist a number of solutions based on this idea, which are tailored specifically to the case $k = 1$. One common approach employs a pair of suffix trees (unmodified and reversed) and a geometric data structure to intersect search results obtained from two suffix trees [Ferragina et al. 1999; Amir et al. 2000].

A recent implementation of this approach [Belazzougui 2009] employs a pair of succinct tries and one-deletion dictionaries (see Section 6.2.10, p. 33). The proposed method uses space $O(\lambda N)$ and answers approximate queries in $O(n + occ)$ time for $k = 1$ in the worst case ($occ$ is the number of occurrences).

In what follows we describe a modification of the reversed dictionary method proposed by Mihov and Schulz [2004]. This modification is based on Corollary 6.2 of partitioning Theorem A.3 (a transposition-aware version is based on Theorem A.6). For simplicity of exposition, we discuss only a transposition-unaware version of the method.

OBSERVATION 6.1. *The restricted Levenshtein distance between strings $p$ and $s$ is equal to the restricted Levenshtein distance between reversed strings:*

$$\mathrm{ED}(p, s) = \mathrm{ED}(\mathbf{rev}(p), \mathbf{rev}(s))$$

Note that Observation 6.1 also holds for Damerau-Levenshtein distance. Observation 6.1 and Theorem A.3 from Appendix A imply:

COROLLARY 6.2. *Let* $\mathrm{ED}(p, s) \le k$ *and* $p$ *be split into two parts:* $p = p_1 p_2$. *Then* $s$ *can be represented as a concatenation of strings* $s_1$ *and* $s_2$ *that satisfy exactly one of the following inequalities:*

$$\begin{array}{lcl}
\mathrm{ED}(p_1, s_1) = 0 & \text{and} & \mathrm{ED}(p_2, s_2) \le k \\
\mathrm{ED}(p_1, s_1) = 1 & \text{and} & \mathrm{ED}(p_2, s_2) \le k - 1 \\
 & \cdots & \\
\mathrm{ED}(p_1, s_1) = \lceil k/2 \rceil - 1 & \text{and} & \mathrm{ED}(p_2, s_2) \le \lfloor k/2 \rfloor + 1 \\
\mathrm{ED}(\mathrm{rev}(p_1), \mathrm{rev}(s_1)) = \lceil k/2 \rceil & \text{and} & \mathrm{ED}(\mathrm{rev}(p_2), \mathrm{rev}(s_2)) \le \lfloor k/2 \rfloor \\
\mathrm{ED}(\mathrm{rev}(p_1), \mathrm{rev}(s_1)) = \lceil k/2 \rceil + 1 & \text{and} & \mathrm{ED}(\mathrm{rev}(p_2), \mathrm{rev}(s_2)) \le \lfloor k/2 \rfloor - 1 \\
 & \cdots & \\
\mathrm{ED}(\mathrm{rev}(p_1), \mathrm{rev}(s_1)) = k - 1 & \text{and} & \mathrm{ED}(\mathrm{rev}(p_2), \mathrm{rev}(s_2)) \le 1 \\
\mathrm{ED}(\mathrm{rev}(p_1), \mathrm{rev}(s_1)) = k & \text{and} & \mathrm{ED}(\mathrm{rev}(p_2), \mathrm{rev}(s_2)) = 0
\end{array}$$

The search procedure uses a pair of path-compressed string tries: one trie indexes the dictionary $W$, while the other indexes reversed dictionary strings: $\{\mathrm{rev}(s) \mid s \in W\}$. We call this data structure ***FB-trie***, which stands for "forward and backward trie".

At search time, the pattern $p$ is split into two parts $p_1$ and $p_2$ of roughly equal lengths (e.g., $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$). Then the method executes a series of $k+1$ two-step sub-queries. There are two types of such sub-queries:

(1) A sub-query of the first type uses the regular trie to find nodes $\xi$ that spell strings $s_1$ such that $\mathrm{ED}(p_1, s_1) = t$ $(0 \le t \le \lceil k/2 \rceil - 1)$. For every node $\xi$ found at the first step, the search procedure recursively traverses descendants of $\xi$ to find leaves that spell strings $s_1 s_2$ that satisfy $\mathrm{ED}(p_2, s_2) \le k - t$.
(2) A sub-query of the second type uses the reversed dictionary trie to find nodes $\xi$ that spell strings $s'_2$ such that $\mathrm{ED}(\mathrm{rev}(p_2), s'_2) \le t'$ $(0 \le t' \le \lfloor k/2 \rfloor)$. For every node $\xi$ found at the first step, the search procedure recursively traverses descendants of $\xi$ to find leaves that spell strings $s'_2 s'_1$ that satisfy $\mathrm{ED}(\mathrm{rev}(p_1), s'_1) = k - t'$. The original dictionary string is equal to $\mathrm{rev}(s'_2 s'_1)$.

Note that there are $\lceil k/2 \rceil$ queries of the first type and $\lfloor k/2 \rfloor + 1$ queries of the second type. Furthermore, threshold values $t$ and $t'$ are upper bounded by

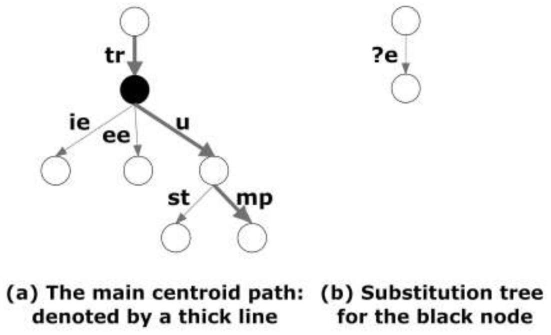
$$\max(\lceil k/2 \rceil - 1, \lfloor k/2 \rfloor) \le \lfloor k/2 \rfloor \le k/2.$$

To verify whether the distance between respective prefixes and suffixes does not exceed $t$ or $t'$, we can use column-wise computation of the dynamic programming matrix (see Section 6.1.1, p. 21), a bit-parallel verification algorithm, or a deterministic Levenshtein automaton (see Section 5).

*6.1.3. $k$-errata tree.* To address the worst-case performance issues that are associated with naive implementations of wildcard and approximate string searching using the trie, Cole et al. [2004] introduced the *$k$-errata tree*. In what follows, we outline a simplified variant of this method that supports only membership queries.

The approach of Cole et al. [2004] blends partial neighborhood generation with the string trie and treats errors by recursively creating insertion, substitution, and deletion subtrees. For transposition-aware searching, transposition trees should also be created. The method uses a centroid path decomposition to select subtrees that participate in the process: the goal of the decomposition is to exclude subtrees with large numbers of leaves and, thus, to significantly reduce the index size. Coelho and Oliveira [2006] proposed a similar data structure: a *dotted suffix tree*, which uses only substitu-

Fig. 6: Example of a centroid path and of a substitution tree. The original trie stores keys: "`tree`", "`trie`", "`trump`", and "`trust`".


tr
ie
ee
u
st
mp
?e
(a) The main centroid path: denoted by a thick line
(b) Substitution tree for the black node

tion subtrees. Unlike the $k$-errata tree, a substitution tree created for a node $\xi$ includes all subtrees of $\xi$.

A *centroid path* starts from a root of a tree. In each node $\xi$, the centroid path branches to a subtree with the largest number of leaves (ties are broken arbitrarily). A subtree of $\xi$ that does not belong to the centroid path is an *off-path* subtree. Each off-path subtree is a starting point of another centroid path. It can be further recursively analyzed to identify centroid paths that start from its descendants. The main centroid path is the one that starts from the root of the entire tree.

Figure 6, Panel (a), illustrates the concept of centroid path decomposition. Consider, for example, the black node, which is a direct descendant of the root. Its rightmost subtree has two leaves, while all other subtrees have only one leaf each. Therefore, the main centroid path branches to the rightmost node. This node has two children, which are both leaves. In this case we have a tie and decide to continue the main centroid path to the right.

To make the search faster, error trees are created: insertion, deletion, substitution, and possibly transposition trees. In addition, error trees that belong to the same centroid path may be merged to form a hierarchy of trees. This allows one to search for the pattern in more than one tree at the same time. We do not discuss all the details of this procedure and address the reader to the paper by Cole et al. [2004] for a complete description.

Instead, we consider an example of a simplified approximate search procedure that allows only substitutions. This procedure relies only on substitution trees (to treat insertions, deletions, and transpositions one needs insertion, deletion, and transposition trees, which are created in a similar way). During indexing, a substitution subtree is created for every *internal* node $\xi$ as follows:

(1) Every off-path subtree of $\xi$ is cloned;
(2) The first character of the subtree root label is replaced by the wildcard character "?" that matches any character;
(3) The modified subtrees are merged: see Panel (b) of Figure 6 for an example;
(4) In a general case of $k > 1$, each error subtree is recursively processed to handle remaining $k - 1$ errors.

Note that the subtree that contains a centroid path does not participate in this process.

The search algorithm starts from the root of the $k$-errata tree and follows the longest path that matches a prefix of the pattern $p$ exactly. Whenever the search exits a centroid path, it has to recursively search all the off-path substitution (insertion, deletion, and transposition) subtrees that hang from this centroid path before the exit point. The search for the longest common prefix stops either:

— On an edge, between two nodes; or
— At an internal node $\xi$.

In the first case, the algorithm considers a possible deletion, insertion, substitution, and transposition. It recursively restarts from either the point of mismatch, or form the point that is one character below. These searches are carried out with the maximum allowed distance equal to $k-1$.

In the second case, there are two sub-searches that should be executed. First, the algorithm considers one edit operation along the centroid path exactly as in the previous case. Second, the search restarts from the root of a substitution (insertion, deletion, and transposition) subtree of the node $\xi$. Again, the maximum allowed edit distance is equal to $k-1$.

To make the search for the longest common prefix efficient, Cole et al. [2004] use an index that perfoms this operation in $O(\log\log\lambda N)$ time, where $\lambda N = \sum_{s\in W}|s|$ is the number of dictionary characters.

Consider a sample pattern “`trze`” and a search procedure that allows at most one substitution (and no other errors). The search procedure matches the prefix “`tr`” and reaches the black node (See Panel (a) of Figure 6). The next pattern character “`z`” does not match any child label. Therefore, we skip one character on the centroid path (denoted by a thick line) and continue searching in the rightmost subtree for an exact match. This search produces no results. Then, instead of considering each of the off-path subtrees separately, the search procedure looks for the pattern suffix “`ze`” in the substitution tree (see Panel (b) of Figure 6). The substitution tree contains the only string “`?e`”, which matches the suffix exactly.

It can be seen that the described simplified variant of the $k$-errata tree fully supports only membership queries, but not regular searches: a single terminal node of an error subtree typically corresponds to several dictionary strings. Furthermore, this variant is not an associative method. A straightforward way to obtain an associative method is to attach a string identifier to each leaf node (which is essentially done in the original method by Cole et al. [2004] employing suffix trees). Then, if the merge procedure combines two leaf nodes $\xi_1$ and $\xi_2$ into the node $\xi$, it also combines identifiers associated with $\xi_1$ and $\xi_2$ into a single list and attaches this list to $\xi$.

The search time of the $k$-errata tree is

$$O\left(n + \frac{(6\log_2 N)^k \log\log\lambda N}{k!} + 3^k \cdot occ\right),$$

where $occ$ is the number of approximate matches in the dictionary. For $k > 1$ and large $N$, the method may be impractical, because the size of the index is upper bounded by

$$O\left(\lambda N + N\frac{(5\log_2 N)^k}{k!}\right)$$

It is possible to reduce the size of the index by the factor of $\log_2 N$ using a suffix-array like structure instead of the trie [Chan et al. 2006].

## 6.2. Neighborhood Generation

The essence of neighborhood generation consists in “reversal” of errors: for instance, if the string “`eta`” is obtained from the string “`beta`” by removal of the first character, it is possible to reverse the error by inserting character “`b`” in the beginning of the string “`eta`”. In general, it is necessary to compute a list of strings obtainable from the search pattern $p$ using at most $k$ edit operations. This list, known as a *full $k$-neighborhood*, consists of the strings $s$ such that $\mathrm{ED}(p, s) \leq k$.

In the classic variant of neighborhood generation method, the search procedure computes the full $k$-neighborhood of $p$ at query time. Then it checks neighborhood members for an exact dictionary match. One of the first implementations of this method is the Unix utility *Ispell* [Kuenning et al. 1988].[16]

In what follows, we discuss methods that employ a compact neighborhood representation: a condensed, super-condensed, and reduced-alphabet neighborhood. Afterwards, we discuss several intermediate methods that memorize deletion-only neighborhoods during indexing time. This discussion is preceded by a description of two neighborhood generation algorithms.

*6.2.1. A Straightforward Algorithm to Generate Full Neighborhood.* A full neighborhood can be constructed by applying various edit scripts containing up to $k$ edit operations. Because these operations are not overlapping (see Section 2.2 for a definition of the restricted edit distance), we need to consider only edit scripts where basic edit operations are ordered by a string position, where these operations are applied. Furthermore, these positions should be strictly increasing.

The straightforward neighborhood generation algorithm can be readily implemented as a recursive procedure with the following input parameters: the current string $s$, the starting position $i_{start}$, and the maximum number of edit operations $k$ that can be applied to $s$ in positions $i \geq i_{start}$.

For example, a process that constructs a one-neighborhood of string `abc` (without transpositions) may have the following steps:

(1) Output original string `abc`;
(2) Output 3 strings `bc`, `ab`, and `ac` obtainable by a single deletion;
(3) Output $4 \times 26$ strings `aabc`, `babc`, ..., `zabc`, `aabc`, `abac`, ..., `azbc`, `abac`, `abbc`, ..., `abzc`, `abca`, `abcb`, ..., `abcz` obtainable by insertion of various alphabet characters at positions one, two, and three as well as at the end of the string.
(4) Output $3 \times 25$ strings `bbc`, `cbc`, ..., `zbc`, `aac`, `acc`, ..., `azc`, `aba`, `abb`, ..., `abz` obtainable by substitution of various alphabet characters for characters at positions one, two, and three.

It can be seen that the algorithm generates 183 strings of which 180 strings are unique and 3 strings (`aabc`, `abbc`, `abcc`) are *repeated twice*.

According to Ukkonen [1993], the total number of edit scripts containing at most $k$ operations and, consequently, the size of the $k$-neighborhood of $p$, can be upper bounded by

$$U_k(p) = \frac{12}{5}(n+1)^k(|\Sigma|+1)^k = O(n^k|\Sigma|^k), \tag{12}$$

where $n = |p|$ and $|\Sigma| \geq 2$ . From the proof of Ukkonen it follows that Equation (12) is also an upper bound for neighborhood generation time and that the same asymptotic upper bound holds for neighborhoods with transpositions.

A disadvantage of the straightforward neighborhood generation method is that it may generate many repeating strings: in the case of DNA data and $k = 3$, the algorithm generates roughly thrice as many strings as the size of the full neighborhood. One approach to solve this problem would be to memorize already generated strings. However, a lookup in the dictionary takes roughly the same time as a lookup in a

[16]Ispell has originated from the Unix utility spell [Gorin 1971], which combines neighborhood generation with partitioning of a dictionary using the first two string letters and string length.

set of already generated strings. Therefore, memorization, which requires extra CPU resources to update the set, only increases retrieval time.

*6.2.2. Neighborhood Generation via Searching in Hypothetical Trie.* A full neighborhood of the pattern $p$ can be obtained by computing the edit distance from $p$ to every string that is at most $|p| + k$ characters long and checking whether the distance is less than or equal to $k$ [Myers 1994]. Because many strings share the same prefix, a more efficient approach is to perform an approximate search in the hypothetical trie that contains all strings no longer than $|p| + k$ characters.

This algorithm can be seen as a recursive parallel traversal of the hypothetical trie and the deterministic Levenshtein automaton, which is presented in Algorithm 1, p. 21. Because the hypothetical trie has a very regular structure, it does not have to be materialized. To simulate the traversal of this trie, Step 3 of Algorithm 1 should be executed $|\Sigma|$ times, setting $v$ to $\Sigma_i$ in the $i$-th iteration. In addition, we should modify Step 1 so that it is executed for every visited node without subsequent backtracking.

Even though this procedure generates only unique strings, in our experiments it is slower than the straightforward neighborhood generation and is feasible only for small alphabets. A modification of this algorithm can be used to efficiently construct a condensed and a super-condensed neighborhood.

*6.2.3. Condensed Neighborhoods.* It is also possible to improve retrieval efficiency of neighborhood generation by eliminating unessential elements. Two types of such elements have been considered in the literature:

(1) Strings that are proper *prefixes* of other neighborhood strings;
(2) Strings that are proper *substrings* of other neighborhood strings;

A neighborhood is called *condensed* [Myers 1994] if it does not contain unessential elements of the first type, i.e., proper prefixes of other strings. A *super-condensed* neighborhood [Russo and Oliveira 2005] does not contain unessential elements of the second type, i.e., proper substrings of other strings. Note that every super-condensed neighborhood is also a condensed neighborhood.

The algorithm to compute a condensed neighborhood is almost identical to the algorithm in Section 6.2.2: it is a recursive parallel traversal of the hypothetical trie over $\Sigma^*$ and the Levenshtein automaton. To exclude strings whose proper prefixes belong to the neighborhood, the algorithm backtracks after the Levenshtein automaton reaches the first accept state.

Russo and Oliveira [2005] proposed a modification of this method that allows one to construct a super-condensed neighborhood. It can be implemented using a pair of the following non-deterministic automata:

— The first automaton recognizes all string prefixes that match the pattern within $k$ errors;
— The second automaton recognizes all substrings that match $p$ within $k$ errors and are not prefixes of $p$.

These automata can be efficiently simulated using the bit-parallel algorithm Shift-AND [Wu and Manber 1992a].

Condensed and super-condensed neighborhoods have been successfully incorporated in approximate *substring* search methods that rely on filtering. A filtering algorithm typically builds an exact substring index of a text string $w$, e.g., a suffix tree. At search time, the pattern $p$ is divided into several pieces: $p = p_1 p_2 \dots p_t$. By generating neighborhoods of pattern pieces $p_i$ and finding all exact occurrences of neighborhood elements in the string $w$, we identify all positions where $p$ may approximately match a

Table II: Average size of a condensed neighborhood as a fraction of a full neighborhood size.

| | $k = 1$ | $k = 2$ | $k = 3$ |
|---|---|---|---|
| Condensed | 0.83 | 0.66 | 0.49 |
| Super-Condensed | 0.68 | 0.41 | 0.19 |

*Note:* DNA Data, one thousand random 11-character sequences.

substring of $w$. Then potential matches are verified by direct comparison with the pattern. This approach – known as pattern partitioning – is discussed in Section 7.1 (see also Theorem A.2, Appendix A).

It can be seen that a set of potential matches associated with unessential neighborhood elements is a subset of potential matches associated with essential neighborhood elements. Therefore, unessential elements could be excluded from consideration.

Exclusion of unessential elements does not lead to a principal modification of the search procedure. However, should we employ a condensed neighborhood for complete string matching and approximate dictionary searching, we would have to replace an exact matching algorithm with a two-step filtering procedure that uses either prefix or substring dictionary searching.

Such filtering method would be slower than a straightforward neighborhood generation unless the size of the super-condensed neighborhood were much smaller than the size of the respective full neighborhood. According to our experimental results, this condition holds only if the pattern is short and the alphabet is small. The difference between the sizes of the super-condensed and the full neighborhood rapidly decreases as the pattern length and/or alphabet size increase. For instance, in the case of DNA sequences of 8 characters, the size of the super-condensed 3-neighborhood is typically 1/30th of the full neighborhood size. For DNA sequences of 11-20 characters that are used in our experiments the difference is at most five-fold (See Table II).

*6.2.4. Wildcard Neighborhood.* The size of a full $k$-neighborhood grows exponentially with the base $n|\Sigma|$ and the exponent $k$, which makes the generation of full neighborhood unfeasible for all but small $k$. The neighborhood can be compressed through excluding strings whose proper substrings belong to the neighborhood, but this method works only for short patterns and small alphabets. In the case of natural languages, better retrieval time can be achieved by using *wildcard neighborhoods*.

Let `?` be a wildcard character that matches any alphabet character. Consider an example of straightforward generation of the one-neighborhood for the pattern `abc`, which generates 183 strings (see Section 6.2.1, p. 26). If we used the wildcard character for insertion and substitution in steps (3) and (4), we would obtain a compact *wildcard neighborhood* that contains only the following 11 strings: `abc`, `?abc`, `a?bc`, `ab?c`, `abc?`, `?bc`, `a?c`, `ab?`, `bc`, `ab`, and `bc`. Generation of the wildcard neighborhood requires little effort. A method that employs the wildcard neighborhood would have short retrieval time, if there were an index that supports wildcard searching efficiently. In the following sections we review two approaches to wildcard indexing.

*6.2.5. Reduced Alphabet Neighborhood Generation.* Consider a hash function $h(c)$ that maps the original alphabet $\Sigma$ to a reduced alphabet $\sigma$ of a smaller size. A character $h(c)$ can be seen as a wildcard that matches every character $a \in \Sigma$ such that $h(a) = h(c)$. The hash function $h(c)$ induces a character-wise projection from $\Sigma^*$ to $\sigma^*$.

The indexing step consists in mapping dictionary strings $s_i$ to strings $h(s_i)$ and indexing $h(s_i)$ for an exact search. Original dictionary strings are stored in buckets based on their projections: if $h(u) = h(v)$, then strings $u$ and $v$ are placed into the same bucket associated with the projection $h(u)$.

During searching, the pattern $p$ is converted into its projection $p' = h(p)$. Then a full neighborhood of $p'$ is generated using $\sigma$. Each element of the neighborhood is searched for exactly. The result of this search is a set of projections and associated dictionary strings.

This step provides a list of candidate strings that should be directly compared with the search pattern $p$. A straightforward implementation of the checking step involves verifying that the Levenshtein distance between $p$ and a candidate string does not exceed the maximum allowed distance $k$.

A more efficient checking approach is to create a wildcard neighborhood of the original pattern using only deletions, transpositions, as well as substitutions and insertions of the character ? that matches any alphabet character. Because the size of the wildcard neighborhood is small, its generation requires little extra time. The elements of the wildcard neighborhood can be efficiently compared with dictionary strings in time proportional to the pattern length. An equivalent verification procedure consists in computation of the Hamming distance. This approach is likely to be a well-known one, but we could not find exact references.

Construction of the wildcard neighborhood is a recursive process synchronized with construction of the reduced-alphabet full neighborhood of $p' = h(p)$. At each step of the recursion, the original pattern $p$ is modified according to the following rules:

— If we delete $p'_{[i]}$, we also delete $p_{[i]}$;
— If we transpose $p'_{[i]}$ and $p'_{[i+1]}$, we also transpose $p_{[i]}$ and $p_{[i+1]}$;
— If we substitute $p'_{[i]}$ with some character $c$, we replace $p_{[i]}$ with ?;
— If we insert a character in position $i$ of $p'$, we insert ? in position $i$ of $p$.

Note that the last two modifications, which introduce the wildcard, apply only to the first insertion and/or substitution in position $i$.

*6.2.6. $k$-deletion Indices.* One neighborhood generation method computes a full $k$-neighborhood of every dictionary string at indexing time. It then indexes neighborhood members for exact matching, which makes retrieval quite fast. However, this method is highly impractical due to enormous space requirements. A better approach relies on index-time memorization of partial neighborhoods. This approach is based on two simple observations, whose justifications follow from the definitions of the restricted edit distance and the optimal alignment (see Section 2.2 and Appendix D.2):

OBSERVATION 6.3. *If* $\mathrm{ED}(p, s) \le k$*, then a wildcard* $k$*-neighborhood of* $p$ *intersects with a wildcard* $k$*-neighborhood of* $s$*.*

OBSERVATION 6.4. *Let* $\mathrm{ED}(p, s) \le k$ *and* $v$ *be a string from the intersection of wildcard* $k$*-neighborhoods of* $p$ *and* $s$*. Let* $v'$ *be a string obtained from* $v$ *by deleting all wildcards. Then* $v'$ *can be obtained from both* $p$ *and* $s$ *using at most* $k$ *deletions.*

These observations immediately give the idea of indexing all strings obtained from dictionary strings by applying $l \le k$ deletions. This approach was proposed by Mor and Fraenkel [1982], though it is better known from the paper by Muth and Manber [1996]. Du and Chang [1994] proposed a similar method, but instead of indexing complete strings obtained by $l \le k$ deletions, they index only specific two-character combinations. As a result, far fewer strings need to be indexed (especially for larger $k$) at the expense of higher verification cost. In what follows, we explain Mor-Fraenkel method for $k = 1$. After that, we discuss the generic case ($k \ge 1$).

*6.2.7. Mor-Fraenkel Method for $k = 1$.* According to Observation 6.4, all single-character mismatches can be detected by indexing deletion-only one-neighborhoods as well as links to respective dictionary strings. In a naive implementation of this idea, the search procedure would generate a deletion-only one-neighborhood of the pattern $p$ and search neighborhood elements for an exact match. For each string found, the search procedure would follow a link to retrieve an original dictionary string.

The naive algorithm is prone to false positives. Let $\Delta_i(s), i > 0$ be an edit operation that removes character $s_{[i]}$ from $s$ and $\Delta_0(s) = s$ be an identity operation. Consider, for instance, the search pattern $p =$ `boxers` and a dictionary containing the string $s =$ `aboxer`. For $k = 1$, the search procedure retrieves $s$, because $\Delta_1(s) = \Delta_6(p) =$ `boxer`. On the other hand, $\text{ED}(p, s) = 2$.

To eliminate false positives, we need to memorize deleted characters and their respective positions. The corresponding indexing algorithm iterates over dictionary strings $\{s \in W\}$, and generates triples $\{(\Delta_j(s), j, s_{[j]})\}$ for $0 \le j \le |s|$. If $j = 0$, then $\Delta_j(s) = s$ and $s_{[j]} = \epsilon$. Then the algorithm indexes triples using $\Delta_j(s)$ as a key (e.g., with the help of hashing or tries). For associative searching, triples can be expanded by adding a pointer to associated data.

Given the pattern $p$, the search procedure generates triples $(\Delta_i(p), i, p_{[i]})$, where $0 \le i \le |p|$. For each triple $(\Delta_i(p), i, p_{[i]})$, the method retrieves the following dictionary triples:

$$\{(s', j, a) \mid s' = \Delta_i(p)\}. \tag{13}$$

Each retrieved triple defines a possibly empty edit script that transforms a dictionary string into $p$:

(1) If $i = j$ and $p_{[i]} = a$, the edit script is empty;
(2) If $i = 0$ and $j \neq 0$, the edit script contains only the deletion of $a$;
(3) If $j = 0$ and $i \neq 0$, the edit script contains only the insertion of character $p_{[i]}$;
(4) If $i = j > 0$ and $a \neq p_{[i]}$, the edit script contains only the substitution of $a$ with $p_{[i]}$;
(5) If $i = j - 1$, $i > 0$ and $a = p_{[i]}$, the edit script contains only the transposition of characters in positions $i$ and $i + 1$;
(6) If both $i$ and $j$ are positive and $i \neq j$, the edit script contains *two* edit operations: $a \to \epsilon$ and $\epsilon \to p_{[i]}$.

It can be seen that every triple $(s', j, a)$ that satisfies Equation (13) and one of conditions 1-5 represents the dictionary string $s$ such that $\text{ED}(p, s) \le 1$. This string is reconstructed from the triple by inserting the character $a$ at position $j$ of the string $s'$. On the contrary, for every dictionary string $s$ that is transformed into $p$ by at most one edit operation, there exist $i$ and $j$ such that $s' = \Delta_j(s)$ and $\Delta_i(p)$ satisfy one of the conditions 1-5.

*6.2.8. Mor-Fraenkel Method: General Case, Transposition-Unaware Searching.* The Mor-Fraenkel method has a straightforward generalization based on generating $l \le k$ deletion-only neighborhoods. First we describe this generalization for transposition-unaware searching and then explain how transpositions can be taken into account.

The indexing algorithm of the generalized Mor-Fraenkel method iterates over dictionary strings $\{s \in W\}$ and generates all possible triples in the form $(s', D^s, C^s)$, where $s' = \Delta_{\tau_l - l + 1}(\ldots \Delta_{\tau_2 - 1}(\Delta_{\tau_1}(s)) \ldots)$ is a string obtained from $s$ by deletion of characters $s_{[\tau_1]}, s_{[\tau_2]}, \ldots, s_{[\tau_l]}$ $(l \le k)$. Positions of deleted characters satisfy $0 < \tau_1 < \tau_2 < \ldots < \tau_l$.[17] $D^s = (\tau_1, \tau_2 - 1, \ldots, \tau_l - l + 1)$ is an $l$-tuple that defines positions of deleted charac-

[17]For simplicity of exposition, we use different notation to explain case $k = 1$ and the generic case. In the case $k = 1$, the unmodified dictionary strings are stored in the form of triple $(s, 0, \epsilon)$. In the generic case, all

ters. $C^s = (s_{[\tau_1]}, s_{[\tau_2]}, \ldots, s_{[\tau_l]})$ is an $l$-tuple (i.e., a list) that stores deleted characters themselves. All triples are indexed using the first element, i.e., $s'$, as a key.

OBSERVATION 6.5. *Given triple $(s', D^s, C^s)$ described above, the string $s$ can be recovered by inserting characters $C^s_i$ at positions $D^s_i + i - 1$ of the string $s'$ in the increasing order of $i$.*

OBSERVATION 6.6. *$D^s$, which encodes positions of deleted characters, is an ordered sequence of possibly repeating integers. Therefore, it is a multiset, i.e., a set that may contain repeated elements.*

The size of the finite multiset $A$ (denoted by $|A|$) is a total number of element occurrences: repeated elements are counted as separate entities. In this survey, we denote finite multisets with integer elements using $l$-tuples, where elements are sorted in ascending order.

Similarly to regular sets, multisets can be represented by a *non-negative integer* function called an *indicator*. The indicator of the multiset $A$ is denoted by $\mathbf{1}_A$. For $e \in A$, $\mathbf{1}_A(e)$ is equal to the number of occurrences of $e$ in $A$. An intersection of multisets can be expressed in terms of the following operation over multiset indicators:

$$\mathbf{1}_{A\cap B}(e) = \min(\mathbf{1}_A(e), \mathbf{1}_B(e)).$$

At query time, the search procedure computes a deletion-only $k$-neighborhood of the pattern $p$ by deleting $m \leq k$ characters in positions $\rho_i$ $(0 < \rho_1 < \rho_2 < \ldots < \rho_m)$. Then it generates triples $(p', D^p, C^p)$, where $p' = \Delta_{\rho_m - m+1}(\ldots \Delta_{\rho_2 - 1}(\Delta_{\rho_1}(p))\ldots)$, $D^p = (\rho_1, \rho_2 - 1, \ldots, \rho_m - m + 1)$ is a *multiset* that defines positions of deleted characters, and $C^p = (p_{[\rho_1]}, s_{[\rho_2]}, \ldots, s_{[\rho_m]})$ is the list that represents deleted characters. Finally, the search procedure retrieves all dictionary triples $(s', D^s, C^s)$ that satisfy the following:

$$p' = s'$$

$$|D^s| + |D^p| - |D^s \cap D^p| \leq k \tag{14}$$

Each triple that satisfies equation (14) represents a dictionary string that matches $p$ within $k$ errors. These dictionary strings can be reconstructed from the retrieved triples.

It can be seen that multisets $D^s$ and $D^p$ define an edit script that transforms $p$ into $s$ using $|D^p| - |D^s \cap D^p|$ deletions, $|D^s| - |D^s \cap D^p|$ insertions, and at most $|D^s \cap D^p|$ substitutions.

Consider an example of strings $s =$`aboxer` and $p =$`boxez`. The edit distance between them is two. To make the strings equal, it is necessary to delete the first and the sixth characters from $s$ as well as the fifth character from $p$. The multisets that represent positions of deleted characters are $D^s = (1, 5)$ and $D^p = (5)$. Note that the second position in $D^s$ is adjusted by the preceding deletion. The intersection contains only one element. Therefore, $|D^s| + |D^p| - |D^s \cap D^p| = 2 + 1 - 1 = 2 = \mathrm{ED}(s, p)$.

As shown in Appendix D.2, if $\mathrm{ED}(p, s) = k$, then $p$ and $s$ satisfy Inequality (14) for some $D^p$ and $D^s$. Therefore, the method allows one to find all strings $s \in W$ such that $\mathrm{ED}(p, s) \leq k$.

*6.2.9. Mor-Fraenkel Method: General Case, Searching with Transpositions.* The indexing algorithm of this transposition-aware variant is described in Section 6.2.8. The retrieval

positions of deleted characters are positive by design, and unmodified dictionary strings are represented by triples $(s, \emptyset, \emptyset)$.

algorithm is essentially the same as that in Section 6.2.8. It also relies on constructing multisets $D^p$ and comparing them with multisets $D^s$ precomputed during indexing. Transpositions are taken into account by correcting $D^p$ for transpositions. For each correction, we insert the modified multiset into Inequality (14) to verify whether $\mathrm{ED}(p, s) \le k$.

Assume that an optimal edit script contains at least one transposition. Then, there exist multisets $D^p$ and $D^s$, which represent deleted positions, as well as lists $C^p$ and $C^s$, which represent deleted characters, that satisfy the following conditions:

$$\begin{array}{l} \exists i, j \;\; D^s_j = D^p_i + 1 \text{ and } C^p_i = C^s_j \\ D^p_i < D^p_{i+1} \text{ or } i = |D^p| \\ D^s_j > D^s_{j-1} \text{ or } j = 1 \end{array} \tag{15}$$

Unfortunately, if the multisets $D^s$ and $D^p$ satisfy the Equation (15), they do not always represent an optimal edit script that contains a transposition. Consider, for instance, strings $s =$"gab" and $p =$"bag". To make the strings equal, one can delete the first and the second characters. The resulting multisets and deletion lists are: $D^s = D^p = (1, 2)$, $C^s = (\mathtt{g}, \mathtt{b})$, and $C^p = (\mathtt{b}, \mathtt{g})$. They satisfy Equation (15) with $i = 1$ and $j = 2$. However, the only optimal alignment consists of two substitutions and does not have a transposition.

To resolve the ambiguity, one should consider all pairs of indices $\{(i_l, j_l)\}$ that satisfy Equation (15). For each pair $(i_l, j_l)$, the *correction for transposition* consists in increasing $D^p_{i_l}$ by one. If the modified $D^p$ satisfies Inequality (14), then the Damerau-Levenshtein distance between $p$ and $s$ does not exceed $k$. Because the Mor-Fraenkel method can be used only with small $k$, the number of corrections is very low in practice. In the worst case this number does not exceed $2^k - 1$ (note that $j_l$ is unambiguously defined by $i_l$). The average number of corrections would be much smaller.

*6.2.10. Transducer-Based Variant of Mor-Fraenkel Method for $k = 1$.* If we use conventional data structures to store deletion-only $l$-neighborhoods, such as hashing tables or tries, the index size can be very large. For instance, an index that satisfies queries for $k \le 2$ is approximately 140 times the size of dictionary (see Table IX, p. 65). To offset exceptional index overhead Mihov and Schulz [2004] proposed to represent $l$-deletion dictionaries ($l \le k$) in the form of *minimal transducers*. For some dictionaries and languages, this method can reduce index size by an order of magnitude. In what follows, we present this idea for $k = 1$. The method can be generalized for $k > 1$, but we omit the description.

A transducer $T(s)$ used in the method of Mihov and Schulz is a deterministic finite state automaton with an output. If $T$ accepts $s$ and, therefore reaches a final state, it outputs a non-empty set of strings $T(s)$. Otherwise, $T(s) = \emptyset$.

Transducers $T_1$ and $T_2$ are equivalent if, for every input $s$, $T_1(s) = T_2(s)$. Among equivalent transducers, Mihov and Schulz choose a transducer with the minimum number of states, i.e., a *minimal transducer*. Minimal transducers can be used for memory-efficient representation of $l$-deletion neighborhoods and can be constructed incrementally [Skut 2004].

The method proposed by Mihov and Schulz [2004], uses the following minimal transducers:

— $T$ is a transducer that accepts only dictionary strings $\{s \in W\}$. If $T$ accepts a string $s$, it also outputs $s$.
— $T_i$ ($i > 0$) accepts all strings $u$ such that $u = \Delta_i(s)$, $s \in W$. In words, it accepts strings obtainable from a dictionary string by deletion of the $i$-th character. If $T_i$ accepts a string $u$, it outputs $\{s \in W \mid u = \Delta_i(s)\}$.

— $T_{\text{all}} = \cup T_i$. It accepts strings $u$ obtainable by a single deletion from a dictionary string (in any position). If $T_{\text{all}}$ accepts $u$, it outputs all strings $s \in W$ such that $u = \Delta_j(s)$ for some $j$.

Given search pattern $p$, the search procedure computes and merges the following sets of strings:

— The pattern (if it is present in $W$): $T(p)$;
— Strings obtainable by a single insertion into $p$: $T_{\text{all}}(p)$;
— Strings obtainable by a single deletion from $p$: $T(\Delta_1(p))$, $T(\Delta_2(p))$, $\ldots$, $T(\Delta_n(p))$;
— Strings obtainable by a single substitution: $T_1(\Delta_1(p))$, $T_2(\Delta_2(p))$, $\ldots$, $T_n(\Delta_n(p))$.

Note that the search procedure may produce duplicates, which have to be eliminated.

It can be also seen that the method described above cannot be used for associative searching directly. To support associative searches, we need an additional index that maps dictionary strings to associated data. Then for every string $s$ found in a transducer-based index, the associated data can be retrieved by searching $s$ in the additional index for an exact match.

The average size of the set $\{s \in W \,|\, \text{ED}(p, s) \le k\}$ is usually small. Furthermore, an exact lookup in the additional index is fast (only a fraction of a microsecond in our PC). Therefore, in many practical situations this additional index supports associative searching with little time and index overhead.

A similar method was proposed by Belazzougui [2009] for $k = 1$. This method uses space $O(\lambda N)$, whereas the hash-based implementation of Mor-Fraenkel method can use up to $\lambda_m^2 N$ bytes ($\lambda_m$ is the maximum length of a dictionary string).

Belazzougui proposed a transducer of a special kind: a *lists dictionary*. Whenever a regular transducer accepts a string $s$ and produces output $T(s)$, an equivalent lists dictionary outputs the same set $T(s)$. However, if the regular transducer does not accept $s$ and produces the empty set, the lists dictionary produces an output $T(s')$, where $s'$ is some string acceptable by the regular transducer $T$. Thus, the Belazzougui transducer can produce false positives, which have to be resolved through direct searching in the dictionary. The dictionary is implemented through minimal perfect hashing (see Section 4.1, p. 16).

In addition to transducers, the method uses a pair of succinct tries to answer queries in $O(n)$ time in the worst case. Belazzougui argues, however, that this data structure is unlikely to be efficient on modern architectures unless it deals with long strings. The rationale is that retrieval times are often dominated by the number of accesses to non-consecutive memory regions (which is large if tries are used).

Therefore, a simplified version of the method that employs the transducers alone may be more efficient in practice. To this end, a transducer $T_i(s)$ can be implemented as a minimal perfect hash index that stores characters $s_{[i]}$ indexed by the string $\Delta_i(s)$. This hash index does not store the strings $\Delta_i(s)$ themselves and uses space $O(\lambda N)$.

Consider a search request that checks for a substitution at position $i$. $T_i(\Delta_i(p))$ represents a set of characters $\{c_j\}$. If there are dictionary strings obtainable from $p$ by a substitution of the $i$-th character, all such strings are obtained from $p$ by replacing $p_{[i]}$ with one of $\{c_j\}$. Otherwise, the characters found represent false positives. These false positives can be discarded by a single search operation that verifies whether the string $p_{[1:i-1]} c_1 p_{[i+1:n]}$ belongs to the dictionary (instead of $c_1$ one can use any single $c_j \in T_i(\Delta_i(p))$).

Instead of a minimal perfect hash function, it is possible to use a regular hash function. The resulting data structure is larger, but it still uses space $O(\lambda N)$. It is also less time-efficient, because a non-perfect hash function can place characters $c_j$ belonging to different strings into a single bucket. Therefore, the search procedure has to verify whether $p_{[1:i-1]}c_j p_{[i+1:n]}$ belongs to the dictionary, for every $c_j \in T_i(\Delta_i(p))$.

*6.2.11. Hybrid Neighborhood Generation.* Another approach to reducing index size consists in combining Mor-Fraenkel method with full neighborhood generation. At indexing time, we use Mor-Fraenkel method to index deletion-only $t$-neighborhoods for $t \leq l < k$. At query time, the search procedure generates full $(k - l)$-neighborhood of the pattern $p$ and searches for its elements in the constructed index with at most $l$ errors.

In our experiments, we have implemented this method for $l = 1$ and $l = 2$: the implementations are denoted by *hybrid neighborhood generation-$l$*. In general, it is possible to combine full query-time neighborhood generation with any approximate search method.

### 6.3. Metric Space Methods

*6.3.1. Metric Space Properties.* Extensive surveys of metric space methods can be found in the paper by Chávez et al. [2001a] and in books by Samet [2005] and Zezula et al. [2005]. These methods require a similarity function $d(x, y)$ that satisfies the three axioms of a metric space:

(1) $d(x, y) = 0 \Leftrightarrow x = y$
(2) $d(x, y) = d(y, x)$
(3) $d(x, y) \leq d(x, z) + d(z, y)$

The third axiom, known as the triangle inequality, is the key property. From the triangle inequality and symmetry (the second axiom) it follows that:

$$d(x, z) + d(y, z) \geq d(x, y) \geq |d(x, z) - d(y, z)| \geq 0 \quad (16)$$

Inequality (16) is the fundamental property of the metric space that allows pruning of the space at query time. To make pruning possible, a metric space method performs a recursive decomposition of the space at indexing time. The obtained hierarchical partitioning of the space is best represented as a tree. A common approach to partitioning involves bucketing: once the number of partition elements falls below bucket size $b$, decomposition stops.

In this review, we focus on dictionary search methods that find all strings within a certain *restricted* edit distance. As we explain in Section 2.2 (see Observations 2.10 and 2.11, p. 9), a straightforward transposition-aware generalization of the dynamic programming algorithm given by Recursion (5) does not produce a subadditive function. This is because the algorithm given by Recursion (5), p. 11, and its weighted generalization given by Recursion (3), p. 10, both compute the *restricted* edit distance, where edit operations do not overlap and do not modify a substring more than once. The restricted edit distance is not always equal to the unrestricted edit distance, but can be computed more efficiently.

Thus, we use computation of the unrestricted Damerau-Levenshtein distance to ensure that metric search methods find all strings within the specified restricted Damerau-Levenshtein distance. To this end, we employ a two-step procedure. First, the search procedure uses the unrestricted Damerau-Levenshtein distance to locate buckets that might contain strings $s$ such that $\text{ED}(p, s) \leq k$. This is achieved through computing distances to *pivots*: specially selected elements of the dataset that are used to partition the dictionary. Whenever the search procedure enters a bucket, it uses an efficient checking procedure (see Section 5, p. 19) to find strings $s$ that satisfy $\text{ED}(p, s) \leq k$ within the bucket. This checking procedure is based on the concept of the restricted edit distance. Because the unrestricted Damerau-Levenshtein distance is a sub-additive lower bound for the restricted Damerau-Levenshtein distance, this method would not miss any true matches.

*6.3.2. Integer-Valued Distance Functions.* The edit distance is an integer-valued function. Not all metric space methods are equally efficient in discrete metric spaces with integer-valued distance functions. In particular, Baeza-Yates and Navarro [1998] argue that the drawback of continuous space methods, such as vantage point trees [Uhlmann 1991], is that they cannot fully use the information obtained from comparison of two data points. This argument is partially supported by experimental evaluation [Fredriksson 2007]. Therefore, we exclude continuous space methods from consideration.

These observations leave us with pivoting methods of two types:

— Recursive pivoting methods;
— Distance-matrix methods.

*6.3.3. Recursive Pivoting Methods.* These methods divide the space with spheres of integer radii and include:

— The *Burkhard-Keller tree* (BKT) [Burkhard and Keller 1973];
— The *Fixed-queries tree* (FQT) [Baeza-Yates et al. 1994];
— The *Fixed-height fixed-queries* tree (FHQT) [Baeza-Yates et al. 1994; Berman 1994];
— The multi-pivot modification of the BKT, which can be seen as a hybrid of the BKT and the FQT.

These methods differ mostly in the approaches to choosing pivots (see Figure 2 on p. 14) and can be seen as variants of the BKT.

In the BKT, a pivot $\pi$ is chosen arbitrarily (in our implementation, randomly). Then dataset $W$ is recursively divided into subsets $W_1$, $W_2$, $\ldots W_l$ such that elements of $W_i$ are at distance $i$ from $\pi$. At query time, the search procedure recursively traverses partitions $i$ that satisfy $|d(\pi, p) - i| \leq k$. The triangle inequality ensures the correctness of this algorithm.

Each BKT node has exactly one pivot. In a multi-pivot modification of the BKT, a node has $t > 1$ pivots [Chávez et al. 2001a]. For instance, if $t = 2$, dataset $W$ is divided into datasets $W_{i,j}$ such that elements of $W_{i,j}$ are at distance $i$ form pivot $\pi_1$ and at distance $j$ from pivot $\pi_2$.

Fixed-query trees, the FQT and the FHQT, are basically BKTs where all nodes of level $i$ share the same pivot $\pi_i$. In the FHQT, all leaves are located at the same level, whereas in the FQT leaves can be located at different levels. Fixed-queries methods have array-based implementations, where running time is traded for smaller index size [Chávez et al. 2001b]. In our experimental setup this tradeoff does not make much sense, because the BKT already has a small index overhead: only a small percentage of the dictionary size (see Table IX, p. 65).

Another modification of the BKT, a parallel BKT (PBKT), was proposed by Fredriksson [2007]. Instead of a single pivot, each tree node stores a bucket of $l$ strings and uses one of them to build the hierarchical space decomposition. At search time, the PBKT uses a bit-parallel algorithm to perform several computations at the same time: the search algorithm computes the exact distance between the search pattern and the pivot as well as verifies whether the edit distance between the search pattern and some of the bucket strings does not exceed $k$.

According to results of Fredriksson [2007], PBKTs are an order of magnitude faster than regular BKTs. However, we believe that the main reason for this improvement is due to use of 128-bit arithmetic and logic operations provided through SSE2 extensions (available on Intel and AMD64 processors). We have not implemented the

PBKT, because comparison of a method that uses 128-bit operations with a bit-parallel baseline method that uses 32-bit operations would not be fair. In addition, enhanced bit-parallelism provided through SSE2 extensions would considerably accelerate implementations of all filtering methods, including unigram frequency vector tries (see Section 7.2.3). Our implementations of string tries and FB-tries rely on bit-parallel simulation of the non-deterministic Levenshtein automaton and, therefore, would improve as well.

*6.3.4. Distance-matrix Methods..* These methods involve computing and storing pairwise distances for a large number of data points. These methods are superlinear in terms of the index size and may be impractical for large datasets. An early distance-matrix method called the *AESA* (Approximating and Eliminating Search Algorithm) was proposed by Vidal [1986]. Its index is simply a matrix that keeps all $N(N-1)/2$ pairwise distances among dictionary strings.

The search algorithm consists in a recursive pruning of the set of candidate strings. At the start, the set of candidate strings encompasses the entire dictionary. At each step, the algorithm chooses a random pivot $\pi$ from the set of candidate strings and measures the distance $\text{ED}(p, \pi)$ from $\pi$ to the pattern $p$. One deterministic approach to choosing a pivot consists in picking up the first element of the candidate set. The search algorithm iterates over all elements in the current set to eliminate strings $s$ that do not satisfy $\text{ED}(\pi, p) - k \le \text{ED}(\pi, s) \le \text{ED}(\pi, p) + k$. Note that all distances $\text{ED}(\pi, s)$ are precomputed and need not be calculated at search time. The search process continues until the candidate set is empty and the query is satisfied. Alternatively, the process can stop when the candidate set contains at most $B > 0$ elements. Then these elements are compared with the pattern directly.

The AESA is $O(N^2)$ in space and construction time that makes it impractical for all but very small dictionaries. There are several modifications of the AESA that achieve AESA-like performance while using much less space by computing either an approximate or a partial distance matrix:

— The *LAESA* stores only $k$ rows of the distance matrix and uses $O(kN)$ space [Micó et al. 1994];
— The *$t$-spanner AESA* employs a $t$-spanner graph to approximate the distance matrix: it stores explicitly only a few distances, while the rest is approximated within a factor $t$. The resulting data structure uses about 3 percent of the size of the original distance matrix [Navarro et al. 2002];
— The *BAESA* approximates the edit distance using $b$ distance intervals. This allows one to approximate the distance matrix using $O(N^2 \log(b))$ *bits* of space [Figueroa and Fredriksson 2007].

Another well-known approach consists in partitioning of the dataset: a dictionary is divided into $m$ parts and each part is indexed using the AESA. At search time, one has to execute $m$ searches in $m$ AESA sub-indices, which can be inefficient. To remedy this problem, Fredriksson [2007] proposed a hybrid of the BKT and the AESA. At the top level, the hybrid behaves exactly as the BKT: it uses arbitrarily chosen pivots to recursively decompose the dictionary into buckets (subsets containing at most $b$ strings). Then it employs the AESA to index the strings inside the buckets. At search time, the method calculates distances to pivots to eliminate the buckets that cannot contain strings satisfying the query. Each bucket that is not discarded in the previous step, is searched using its own AESA index.

There exist a number of metric space methods that do not guarantee retrieval of all strings within distance $k$ (see, e.g., [Chávez and Navarro 2003]). These methods, however, are out of the scope of this survey.

## 7. INDEXING METHODS BASED ON SEQUENCE FILTERING

### 7.1. Pattern Partitioning

An early description of a pattern partitioning method can be found in Knuth [1997], p. 394.[18] Knuth noted that an original word and its slightly misspelled version probably agreed in the first or in the second part. One of the first implementations of this idea is described by Doster [1977].

Indexing of string fragments has been commonly used already in the 70s and 80s for the purpose of document retrieval [Schuegraf 1973; Schek 1978; Willet 1979] and approximate dictionary searching [Angell et al. 1983]. These early works rely mostly on common substring similarity measures (based on the number of common substrings, e.g., $q$-grams). Jokinen and Ukkonen [1991] describe one of the first pattern partitioning methods that relies on the Levenshtein distance.

Classic partitioning algorithms search for pattern parts (features) in dictionary strings. The most commonly used features are:

(1) regular $q$-grams: contiguous substrings of the same length $q > 1$;
(2) unigrams: $q$-grams of size one (individual characters);
(3) multi-grams: contiguous substrings of variable length [Schek 1978; D'Amore and Mah 1985; Cho and Rajagopalan 2002; Navarro and Salmela 2009];
(4) $q$-samples: non-overlapping $q$-grams [Sutinen and Tarhio 1996; Navarro et al. 2001; Navarro et al. 2005];
(5) gapped $q$-grams: non-contiguous string subsequences [Burkhardt and Kärkkäinen 2002].

For higher filtering efficiency, features are usually enhanced with information on their positions in a string. In particular, one can use *positional $q$-grams*, i.e., pairs $(s_{[\rho:\rho+q-1]}, \rho)$.[19]

There are two common approaches to pattern partitioning: *partitioning into exact searching* and *intermediate partitioning*. Partitioning into exact searching implies that features are searched for exactly. Intermediate partitioning involves approximate feature searching. One variant of intermediate partitioning consists in searching for parts of dictionary strings in the pattern. This method uses an index of dictionary $q$-samples.

In what follows, we review these approaches. These rely on several pattern partitioning properties that are only outlined here. A more detailed discussion, which includes the case of transposition aware searching, is presented in Appendix A.

We also discuss possible implementations, focusing on partitioning into exact searching and $q$-gram indexing.

#### *7.1.1. Partitioning into Exact Searching.*

Partitioning into exact searching is based on the following well-known observation (see Theorem A.2, Appendix A):

OBSERVATION 7.1. *Let* $\mathrm{ED}(p, s) \leq k$. *If we divide the string* $p$ *into* $k + 1$ *substrings* $p_1, p_2, \ldots, p_{k+1}$*, then at least one* $p_i$ *is an exact substring of* $s$*. The position of the matching substring differs from its original positing in* $p$ *by at most* $k$*.[Wu and Manber 1992b].*

A transposition-aware modification is based on Theorem A.5. For simplicity of exposition, we explain only the transposition-unaware method.

Given the pattern $p$ and the maximum allowed edit distance $k$, the search procedure splits $p$ into $k + 1$ contiguous substrings $p_1, p_2, \ldots, p_{k+1}$ starting at positions $\rho_1$, $\rho_2$, ..., $\rho_{k+1}$ ($\rho_1 = 1$). Then it retrieves all dictionary strings that contain at least one $p_i$ as an

[18] The first edition of the book was published in 1973.
[19] See, e.g., [Riseman and Hanson 1974], where positional $q$-grams are used for spell-checking without a dictionary.

exact substring starting at position $\tau$ such that $|\tau - \rho_i| \le k$. An optimal partition can be determined by using statistics of substring occurrences [Navarro and Baeza-Yates 1998; Navarro and Salmela 2009].

The crucial step of the search procedure is efficient retrieval of dictionary strings that contain a specified substring. It can be implemented with the help of a suffix-tree-like data structure (see Section 4.3). In this survey, we use a simpler method: an inverted $q$-gram file. The inverted $q$-gram file keeps a list of $q$-grams that occur in dictionary strings. The list of $q$-grams is indexed for faster access. For each $q$-gram, the inverted $q$-gram file keeps a list of pointers to strings where this $q$-gram occurs. Such an occurrence list is an *inverted list*.

A bit-sliced *signature file* is an alternative lossy representation of an inverted file, where inverted lists are stored as bit vectors, i.e., signatures, and inverted lists of different $q$-grams can be OR-ed together. Although for the purpose of document retrieval, the signature file has been shown to be inferior to the inverted file [Zobel et al. 1998], there is no consensus on which data structure is better for approximate dictionary matching. Zobel and Dart [1995] have stated that inverted files had “markedly better performance” (but no detail is given). On the other hand, Carterette and Can [2005] have demonstrated that the signature file has equivalent retrieval times and smaller index size.

*7.1.2. Inverted $q$-gram file.* To construct an inverted $q$-gram file, an indexing procedure iterates over the dictionary strings $s$ and pads each $s$ with $q-1$ additional spaces on the right (padding guarantees that, given a string $u$ such that $|u| < q$, it is possible to retrieve all strings $s$ containing $u$ as an exact substring). Then the indexing procedure computes the list of positional $q$-grams of the padded string $s'$. Recall that a positional $q$-gram is a pair $(s_{[i:i+q-1]}, i)$, where $i \le |s| - q + 1$ is the starting position of $q$-gram $s_{[i:i+q-1]}$).

Positional $q$-grams are indexed using an inverted file. Given a positional $q$-gram $(u, i)$, its inverted list provides pointers to strings $s'$ such that $s'_{[i:i+q-1]} = u$. To improve running time, we also divide the inverted file into sub-indices, where each sub-index corresponds to strings of the same length.

At search time, pattern $p$ is split into $k+1$ fragments $p_1, p_2, \ldots, p_{k+1}$ starting at positions $\rho_1, \rho_2, \ldots, \rho_{k+1}$ ($\rho_1 = 1$). For each $p_i$, the method uses the inverted file to retrieve a list of pointers to strings that may contain $p_i$ as an *exact* substring at positions $\tau$ such that $|\rho_i - \tau| \le k$. Then the search procedure merges pointer lists obtained and removes duplicates. The result is the list of candidates for the checking step of the method. At the checking step candidate strings are compared with the search pattern element-wise. Note that the $q-1$ spaces added at the indexing step are not used in this comparison.

In what follows, we explain the substring search procedure in more detail focusing on $q$-gram based implementation. Strings $\{s \in W\}$ that contain $p_i$ as an exact substring are easily found when $|p_i| = q$ (the length of substring $p_i$ is equal to the length of $q$-gram).

Let $|p_i| > q$ and $\{v_l\}$ be the set of $|p_i| - q + 1$ $q$-grams contained in $p_i$. Note that $v_l$ starts at position $\rho_i + l - 1$. For each $v_l$, the search procedure retrieves inverted lists of positional $q$-grams $(v_l, \tau_l)$ such that $|\rho_i + l - 1 - \tau_l| \le k$ (the inverted lists contain only word identifiers and do not keep information on $q$-gram positions). Then we compute the union of these lists, which is denoted by $U_{i,l}$. It can be seen that the intersection $\cap_l U_{i,l}$ contains pointers to all strings that have $p_i$ as an exact substring (starting at position $\tau$ such that $|\rho_i - \tau| \le k$). Yet, this intersection may include a few strings that do not contain $p_i$ as an exact substring (because positions of $q$-grams $v_l$ do not exactly match positions inside dictionary strings).[20] Such strings are eliminated by direct comparison with the pattern. To save processing time, we can stop intersecting lists $U_{i,l}$ once the size of the intersection falls below a threshold [Zobel and Dart 1995], because the cost of checking for an approximate match can be lower than the cost of list processing. Alternatively, the search procedure may intersect only $M$ lists $U_{i,l}$, where $M$ is a parameter chosen empirically.

If $|p_i| < q$, the search procedure processes inverted lists of positional $q$-grams $(p_i v, \tau)$ such that $|\rho_i - \tau| \leq k$. In words, it retrieves inverted lists of $q$-grams that have prefix $p_i$ and start at position close to $\rho_i$. The union of these lists contains pointers to all dictionary strings that include $p_i$ as an exact substring (at position $\tau$ such that $|\tau - \rho_i| \leq k$).

It is common to keep entries in inverted $q$-gram lists sorted by respective string identifiers. Then both unions and intersections of inverted lists $S_1$ and $S_2$ can be computed using a merging algorithm in time proportional to $|S_1| + |S_2|$, where $|S|$ is the number of elements in list $S$ (see, e.g., [Knuth 1997], p. 158). If one inverted list is much longer than another, it is possible to compute an intersection more efficiently, e.g., by binary searching for the elements of the shorter list in the longer list. In general, a shorter running time can be achieved by applying an algorithm that adapts to input values [Demaine et al. 2001; Baeza-Yates and Salinger 2005]. See also [Barbay et al. 2006; Claude et al. 2009] for a description of efficient intersection algorithms.

If the pattern has fewer than $k+1$ characters, it cannot be split into $k+1$ non-empty substrings. Formally, the pattern partitioning lemma is not applicable, if $n < k+1$. Furthermore, the strings shorter than $2k+1$ characters are not retrievable after $k$ deletions are applied (because respective search patterns would have fewer than $k+1$ characters). Consider, for instance, $s =$`ox`, $p =$`x`, and $k = 1$.

Therefore all strings shorter than $2k+1$ characters have to be stored in a separate index (in addition to the main $q$-gram index). This auxiliary index is used to satisfy queries when $|p| < k+1$. Otherwise, the main index is used.

For small $k$, we can store short strings as a plain file and search this file sequentially. Sequential search is executed infrequently, only when $|p| < k+1$. In our implementation, we use $k$ auxiliary plain files: the $i$-th file contains strings of lengths from $2k-1$ to $2k$. Another option would be to combine an inverted $q$-gram file with neighborhood generation. Because we search for patterns shorter than $k+1$ characters, this approach is efficient.

*7.1.3. Intermediate Partitioning.* Intermediate partitioning is based on Theorem A.2 and Theorem A.5, see Appendix A, p. 76. For simplicity of exposition, we outline only the transposition-unaware method based on Theorem $A.2$.

At query time, the pattern $p$ is split into $j \geq 1$ fragments $p_i$, starting at positions $\rho_i$. These fragments are used to find dictionary strings $s$ such that

$$\mathrm{ED}(s_{[\tau_1:\tau_2]}, p_i) \leq \lfloor k/j \rfloor$$

for some $\tau_1$ satisfying $|\rho_i - \tau_1| \leq k$. The result of approximate substring searching is a list of candidate strings, which are compared with the pattern $p$ element-wise. An important step is to find dictionary strings that contain $p_i$ as an approximate substring. There are three common approaches for accomplishing this task.

[20]Consider, for instance, $p =$`potential`, $q = 2$, and $k = 1$, The above-described method will retrieve the string $s =$`posterior`, because the latter contains $q$-grams “`po`” and “`te`” starting in positions 1 and 3. Yet, the string “`posterior`” does not have the substring “`pote`”.

The first approach employs the suffix tree (see Section 4.3). Approximate searching is carried out using the backtracking algorithm described in Section 6.1.1. The same algorithm can be simulated over the suffix array [Navarro and Baeza-Yates 2000].

In the second approach, approximate substring searching is reduced to exact searching by generating a neighborhood of $p_i$. The neighborhood can be full, condensed [Myers 1994], or super-condensed [Russo and Oliveira 2005]. The advantage of this approach is that the neighborhood of $p_i$ can be much smaller than the full pattern neighborhood.

Dictionary strings containing neighborhood elements as exact substrings can be retrieved with the help of the inverted $q$-gram file as described in the Section 7.1.2. Note that intermediate pattern partitioning methods do not need an additional index to handle short patterns.

For $j = 2$, there exists an important variant of intermediate pattern partitioning theorem that we discuss in Section 6.1.2. A very efficient method – FB-trie – is based on this theorem.

The third approach searches for dictionary $q$-samples inside the pattern. It is discussed in Section 7.1.4.

*7.1.4. $q$-sampling.* The idea of using $q$-samples for on-line searching was suggested by Sutinen and Tarhio [1996]. Based on this idea, Navarro et al. [2005] proposed an indexing method for approximate substring searching. In what follows, we discuss a modification of this method tailored to (transposition-aware) dictionary searching. It consists in indexing every $h$-th $q$-gram ($h \geq q$) of dictionary strings. Such non-overlapping $q$-grams are called *$q$-samples*. One can see that the resulting index requires less space than the index that keeps all $q$-grams of dictionary strings.

Formally, for every dictionary string $s$, we collect $j = \lfloor |s|/h \rfloor$ substrings $s_{[1:q]}$, $s_{[h+1:h+q]}$, ..., $s_{[(j-1)h+1\,:\,(j-1)h+q]}$ and index them using the inverted file. The list of the unique $q$-samples, i.e., the dictionary of the inverted file, is stored as the trie.

In the case of transposition-aware searching one should choose $h > q$. Otherwise, a single transposition can modify two $q$-samples (see Observation 7.5 below). An alternative solution for the case $h = q$ consists in indexing additional $j - 1$ $q$-samples. The $i$-th additional $q$-sample is obtained from the $i$-th original $q$-sample $s_{[(i-1)q+1\,:\,iq]}$ by replacing its last character with $s_{[iq+1]}$.

The search algorithm identifies dictionary strings containing $q$-samples that match a substring inside the pattern approximately. This step provides a list of candidates that are directly compared with the search pattern. Below, we discuss this algorithm in detail.

OBSERVATION 7.2. *Let $\mathrm{ED}(p, s) \leq k$ and $u$ be a substring of $s$. Then, there is a substring $v$ of $p$ such that $\mathrm{ED}(u, v) \leq k$.*

This observation immediately follows from the definition of the optimal alignment (see Section 2.2) and holds for both the Levenshtein and the Damerau-Levenshtein distance.

THEOREM 7.3. *Let $\mathrm{ED}(p, s) \leq k$ and the string $s$ be partitioned into $2j$ contiguous (possibly empty) substrings: $s = s_1u_1s_2u_2 \ldots s_ju_j$. Then, the string $p$ contains $j$ substrings $p_i$ such that*

$$\sum_{i=1}^{j} \mathrm{ED}(p_i, s_i) \leq k \qquad (17)$$

COROLLARY 7.4. *The sum of $j$ non-negative terms $\mathrm{ED}(p_i, s_i)$ in Inequality (17) is at most $k$. Therefore, there exists $l$ such that $\mathrm{ED}(p_l, s_l) \leq \lfloor k/j \rfloor$.*

In the case of the Levenshtein distance, Theorem 8.3 follows from Lemma A.1 (p. 75) and Observation 7.2. The proof consists in applying edit operations (in increasing order of modified substring positions) and registering the changes.

OBSERVATION 7.5. *In the case of the Damerau-Levenshtein distance, Theorem 7.3 holds only if $u_i \neq \epsilon$ for all $i < j$.*

To see why this observation is true, consider an example of the string $s =$abcd partitioned into substrings $s_1 =$ ab and $s_2 =$ cd ($u_1 = u_2 = \epsilon$). Consider also the string $p =$acbd obtainable from $s$ by one transposition. Note that both $s_1$ and $s_2$ match any substring inside $p$ with one or two errors. Therefore, the sum in Inequality (17) is at least 2, whereas $k = \text{ED}(p, s) = 1$.

Let $\rho_i$ be the position of the substring $s_i$ in $s$. We divide all insertions and deletions into two groups. There are $k' \le k$ insertions/deletions applied before the position $\rho_i$ and at most $k - k'$ insertions/deletions applied at or after the position $\rho_i$. As a result of applying $k'$ edit operations from the first group, the position of $s_i$ cannot increase or decrease by more than $k'$. After applying operations from the second group, the length of $s_i$ cannot increase by more than $k - k'$. Therefore, the following observation is valid:

OBSERVATION 7.6. *A substring of $p$ that approximately matches $s_i$ starts at or after the position $\rho_i - k$ and ends at or before the position $\rho_i + |s_i| + k - 1$.*

Assume now that we have an inverted file of $q$-samples and $s$ is a dictionary string such that $\text{ED}(p, s) \le k$. Because $|s| \ge |p| - k = n - k$, the inverted file should contain at least $j = \lfloor (n - k)/h \rfloor$ $q$-samples from $s$. According to Theorem 7.3 and Observation 7.6, each $q$-sample of $s$ matches a substring of $p$ with at most $k$ errors. Note that the $i$-th $q$-sample of $s$ has a match within the pattern substring $w_i = p_{[(i-1)h+1-k\,:\,ih+k]}$. Furthermore, the sum of edit distances between $q$-samples and respective approximate matches inside $p$ should not exceed $k$. This sum is represented by Inequality (17).

Thus, to answer an approximate dictionary query, one can retrieve strings satisfying Inequality (17). This can be achieved through computing a lower bound for $\text{ED}(p, s)$, termed as a *cumulative best match distance*. This lower bound is equal to the sum of the best (i.e., minimum) edit distances between consecutive $q$-samples of the string $s$ and substrings within $w_i$:

$$\sum_{i=1}^{j} \text{BED}(w_i, s_{[(i-1)h+1\,:\,(i-1)h+q]}), \tag{18}$$

$$\text{where } \text{BED}(w, u) = \min_{1 \le i_1 \le i_2 \le |w|} \text{ED}(w_{[i_1:i_2]}, u).$$

Navarro et al. [2005] proposed to use $w_i = p_{[(i-1)h+1\,:\,ih+q-1+k]}$, but we omit the justification. Because the pattern substrings are extended only to the right (as opposed to extending each pattern piece by $k$ characters in both directions), this approach does not guarantee that the $i$-th $q$-sample is necessarily aligned with $p_i$ in Equation (18). More specifically, there exists $t \ge 0$ such that $\sum_{i=1}^{j} \text{BED}(w_i, s_{[(t+i-1)h+1\,:\,(t+i-1)h+q]}) \le k$.

To evaluate the cumulative best match distance, one may retrieve all $q$-samples that occur as approximate pattern substrings with at most $k$ errors. A more practical approach is to estimate the cumulative best match distance, rather than compute it exactly. This estimate should be a lower bound for $\text{ED}(p, s)$.

To this end, we only retrieve $q$-samples that match substrings inside $w_i$ with at most $k_e$ errors, where the parameter $k_e$ satisfies $\lfloor k/j \rfloor \le k_e \le k$. Additionally, we require that $k_e < q$. Otherwise, every $q$-sample is retrieved, because it matches a substring of $p$ with $q$ errors.

The choice of $k_e$ involves a tradeoff: the smaller $k_e$ is, the less time is spent on computing the lower bound. On the other hand, smaller values of $k_e$ lead to "looser" lower bounds and, therefore, entail higher verification costs.

The search algorithm is as follows. Assume that every dictionary string is associated with $j$ variables $B_i$. The initial values of the variables are equal to $k_e + 1$. The search procedure uses the trie to identify all $q$-samples that have a match inside the pattern substrings $w_i$ with at most $k_e$ errors.

Let the $q$-sample $u$ be an approximate substring match of $w_i$ with $k'$ errors. Then, for every dictionary string such that its $i$-th $q$-sample is equal to $u$, the variable $B_i$ is reset to $\min(B_i, k')$. If the sum of variables assigned to the string $s$ becomes smaller than $k + 1$, we add $s$ to the list of candidates for subsequent verification.

From Corollary 7.4 and the condition $k_e \geq \lfloor k/j \rfloor$, it follows that every dictionary string $s$ such that $\mathrm{ED}(p, s) \leq k$ contains a $q$-sample that matches a substring of $p$ within $k_e$ errors and, thus, will be examined by the algorithm. Note that it is not necessary to maintain $B_i$ for every dictionary string: the algorithm has to allocate variables $B_i$ for the string $s$ the first time it encounters a $q$-sample of $s$.

The parameter $k_e$ satisfies: $\lfloor k/j \rfloor \leq k_e < q$, which implies that $k/j \leq q$. Because $q \leq h$ and $j \leq (n - k)/h$, the method is applicable only if $n$, $h$, and $k$ satisfy:[21]

$$\frac{hk}{n-k} \leq h \Rightarrow n \geq 2k. \qquad (19)$$

From Inequality (19) it follows that, similar to the inverted $q$-gram file, the method is not applicable for short patterns.

## 7.2. Vector Space Frequency Distance Methods

Vector space search methods can be seen as variants of pattern partitioning methods, which involve conversion of features (usually $q$-grams and unigrams) into *frequency vectors* or *signatures*. These methods have been often discussed in the literature. In particular, the idea of using string signatures to filter out strings that do not match the search pattern was proposed by Damerau [1964]. Unigram and $q$-gram signatures have been extensively used for "dictionaryless" spell-checking in the 70s, see the review by Kukich [1992] and references therein. Vector space methods are also employed in computational biology [Ozturk and Ferhatosmanoglu 2003; Kahveci et al. 2004; Wang et al. 2009].

In this survey, we focus on methods that use frequency distance (see Section 2.3 for a definition of frequency distance). A number of papers explore the possibility of using the Hamming distance in place of frequency distance, see, e.g., [Giladi et al. 2002; Gollapudi and Panigrahy 2006]. Given strings $p$, $s$, and their *unigram* frequency vectors $\mathrm{vect}(p)$ and $\mathrm{vect}(s)$, the Hamming distance between $\mathrm{vect}(p)$ and $\mathrm{vect}(s)$ can be *twice the edit distance* between $p$ and $s$,[22] whereas the frequency distance between $\mathrm{vect}(p)$ and $\mathrm{vect}(s)$ is *always a lower bound* for the edit distance between $p$ and $s$. Because frequency distance is a better estimate of the edit distance than the Hamming distance, we discuss and implement only frequency distance based methods.

Given a projection of strings into a low-dimensional vector space, we substitute original string queries (with the maximum allowed edit distance $k$) by *frequency distance* queries in the projected space. According to Inequality (11), p. 13, the maximum al-

[21]For transposition-aware search, one has to index artificial $q$-samples or to choose $h \geq q + 1$.

[22]Consider, for instance, a single substitution in a string $s$ that modifies exactly two elements of the frequency vector $\mathrm{vect}(s)$.

lowed frequency distance in the projected space is

$$k \cdot (q + [\text{transposition-aware and } q > 1])$$

This depends on the maximum allowed edit distance $k$, the size of $q$-gram $q$, and the type of query (transposition-aware or transposition-unaware searching).

For efficient retrieval, frequency vectors or signatures are indexed using general vector space methods. Dimensionality is reduced via hashing: a hash function is used to map the original alphabet $\Sigma$ to a reduced alphabet $\sigma$ of a smaller size (see a discussion in Section 7.2.2)

Dictionary strings are grouped into buckets based on their frequency vectors. For each vector $x$, a vector space data structure stores a pointer to the bucket that contains strings mapped to the frequency vector $x$.

The filtering step gives a list of candidate strings, which are compared with the pattern element-wise using an efficient on-line checking algorithm (outlined in Section 5, p. 19). Because $\text{FD}(\text{signat}(p), \text{signat}(s)) \leq \text{FD}(\text{vect}(p), \text{vect}(s))$, the signature-based search is carried out with the same threshold.

A string can be transformed into either a unigram or a $q$-gram sequence, which can be then converted into a frequency vector or a signature. Therefore, there are four scenarios of mapping to vector space, which involve transformation of strings into:

(1) unigram frequency vectors;
(2) $q$-gram frequency vectors ($q > 1$);
(3) unigram signatures;
(4) $q$-gram signatures ($q > 1$).

We have implemented methods for scenarios 1, 2, and 3. In what follows, we briefly overview these implementations as well as respective vector space data structures and dimensionality reduction methods.

*7.2.1. Space Decomposition Tree.* To efficiently satisfy frequency distance queries, we use data structures designed for searching in multidimensional spaces. Based on their approach to data organization, multidimensional search methods can be classified into *space-partitioning* methods and *data-partitioning* methods (see, e.g., [Weber et al. 1998]).

Space-partitioning search methods recursively decompose space into mutually disjoint parts (usually along predefined hyperplanes). They include the KD-tree [Bentley 1975; Friedman et al. 1977], the K-D-B-tree [Robinson 1981], the KD-trie [Orenstein 1982], and the grid file [Nievergelt et al. 1984], to name a few.

Data-partitioning search methods divide data rather than space. These methods include the RD-tree [Hellerstein and Pfeffer 1994], the R-tree [Guttman 1984] as well its modifications such as the R$^{+}$-tree [Sellis et al. 1987] and the R*-tree [Beckmann et al. 1990]. For a detailed discussion we refer the reader to the surveys by Gaede and Günther [1998], Böhm et al. [2001], as well as to the books by Faloutsos [1996] and Samet [2005].

Data partitioning methods divide a dataset using bounding rectangles or spheres that often have overlapping regions. These are essential only for spatial data, e.g., for polygons and lines, but not for point data. As noted by Roussopoulos and Leifker [1985], Samet [1995], and others, these overlaps often result in poor performance. Therefore, we concentrate on space partitioning methods.

We employ a *vector trie* and the *KD-tree*, because they are efficient and easy to implement. Both methods were described in and implemented for the conference version of this survey [Boitsov 2004], but its likely that earlier references exist. In particular, there are a number of similar methods. Mochizuki and Hayashi [2000] suggested to index $q$-gram signatures in a trie to answer exact substring queries. Ozturk and Ferhatosmanoglu [2003] as well as Jeong et al. [2010] indexed $q$-gram frequency vectors using R-trees. Wang et al. [2009] proposed to combine the vector trie with the inverted $q$-gram file.

The vector trie is the path-compressed trie (See Section 4.2) that is built on coordinate vectors. The KD-tree [Bentley 1975; Friedman et al. 1977] is a binary space decomposition tree that uses hyperplanes perpendicular to one of the coordinate axes. Every non-leaf node $\xi$ is associated with a bounding rectangle and a hyperplane that divides this rectangle into two parts. Each part represents a child node of $\xi$. The dividing hyperplane is defined by equation $x_d = a$, where $d$ is a *discriminating dimension*. There are several approaches to choosing $d$. One approach is to choose $d$ in a cyclic manner. The KD-tree may degenerate to linear searching and requires periodic rebalancing to ensure efficient retrieval. For semi-static dictionaries, however, balancing costs are spread over multiple retrieval and update operations.

It can be seen that both KD-trees and vector tries belong to the class of methods that divide the space with axis-orthogonal hyperplanes. An indexing algorithm of such method starts from the whole space and divides it into bounding (hyper) rectangles that do not intersect except along edges. Then the indexing algorithm proceeds recursively on each of the bounding rectangles until it reaches a desired level of granularity. The result is a hierarchical decomposition of space into nested bounding rectangles, which is best represented as a tree.

A search algorithm recursively traverses the tree and discards all the nodes $\xi$ such that a frequency distance between vect($p$) (pattern frequency vector) and the bounding rectangle of the node $\xi$ is above a search threshold.

According to results of preliminary experiments, all of our trie-based implementations are slightly more efficient than KD-tree-based implementations. Therefore, we index frequency vectors using tries.

*7.2.2. Dimensionality Reduction.* It is well-known that vector space search methods perform poorly in high-dimensional spaces [Weber et al. 1998; Chávez et al. 2001a]. This phenomenon is known as the “curse of dimensionality”. Weber et al. [1998] have shown that every nearest neighbor search method degrades to sequential searching if the number of dimensions is sufficiently large. In particular, R*-trees [Beckmann et al. 1990] and X-trees [Berchtold et al. 1996] are outperformed by a linear scan when the number of dimensions is greater than 10, and by a vector approximation file (VA-file) when the number of dimensions is greater than 6 [Weber et al. 1998].

The size of a frequency vector is equal to the size of the alphabet. Because all major natural language alphabets have more than 20 characters, the curse of dimensionality poses a serious problem for indexing both $q$-gram and unigram frequency vectors. To decrease dimensionality, we use a commonly known method that involves hashing. It consists in projecting of the original alphabet $\Sigma$ to an alphabet $\sigma$ of a smaller size using a hash function $h(c)$.

The hash function $h(c)$ induces a character-wise projection from $\Sigma^*$ to $\sigma^*$. Given a string $s$, vect($h(s)$) is a *reduced frequency vector* of $s$. Recall that the size of a string frequency vector is equal to the size of the alphabet. Therefore, the size of reduced frequency vector vect($h(s)$) is smaller than the size of original frequency vector vect($s$) (because $|\sigma| < |\Sigma|$). Further, we note that for any two strings $p$ and $s$, a frequency distance between reduced frequency vectors is also a lower bound for the edit distance between $p$ and $s$:

$$\mathrm{FD}(\mathrm{vect}(h(p)), \mathrm{vect}(h(s))) \le \mathrm{ED}(h(p), h(s)) \le \mathrm{ED}(p, s).$$

In addition, this inequality holds if we substitute $p$ and $s$ with arbitrary $q$-gram sequences, which are strings over $\Sigma^q$.

Consequently, we can use hashing to map data points from a high-dimensional vector space, where the number of dimensions is $|\Sigma|^q$, to a low-dimensional vector space, where the number of dimensions is $|\sigma|$ (the size of the reduced alphabet). We can then execute frequency distance queries in the space of reduced frequency vectors using the same threshold value as in the original vector space. A brief description of hash functions used in our experiments is given in Appendix B.

*7.2.3. Unigram frequency vector trie.* A unigram frequency vector trie uses conversion of strings to unigram frequency vectors, which are treated as strings over a finite alphabet. These strings are indexed using the path-compressed trie (see Section 4.2). Let $z = \text{vect}(p)$ be the frequency vector of the search pattern $p$.

The search procedure is a recursive traversal of the trie. Whenever it reaches a node $\xi$, which spells vector $u$ of the length $l$, the frequency distance between $u$ and the prefix of $z$ of the length $l$ has been calculated. It can be seen that this value represents the frequency distance between $\text{vect}(p)$ and the bounding rectangle that corresponds to the node $\xi$. Recursion continues while this distance does not exceed the maximum allowed distance $k$.

The result of this search is a list of frequency vectors $x$ such that $\text{FD}(x, z) \leq k$. For each vector found in the trie, the search procedure retrieves the list of respective strings and compares them with the search pattern.

*7.2.4. $q$-gram frequency vector trie.* A $q$-gram frequency vector trie employs conversion of strings to $q$-gram frequency vectors ($q > 1$). Then, similarly to the unigram frequency vector trie, $q$-gram frequency vectors are stored using the path-compressed trie (see Section 4.2). The only difference is that, at search time, the $q$-gram frequency vector trie a uses larger threshold value that is equal to (see Section 2.3.3, p. 13):

$$k \cdot (q + [\text{transposition-aware}]).$$

*7.2.5. Signature hashing.* This simple folklore approach employs conversion of strings into small unigram signatures. Dictionary strings are stored as a hash table. A string signature is essentially an integer value, which is used as a hash table slot number. Collisions in the hash table are resolved via chaining: a slot in the hash table that corresponds to a signature $x$ contains a pointer to the list of strings $\{w_i\}$ such that $\text{signat}(w_i) = x$.

At search time, the method scans hash table slots to find all signatures $x$ such that $\text{FD}(x, z) \leq k$, where $z = \text{signat}(p)$ is the signature of the search pattern. To make sequential scan efficient, we choose the signature size $m$ so that the total number of signatures $2^m$ is small in relation to the number of dictionary strings. The optimal value of $m$ is determined experimentally.

## 8. EXPERIMENTS

### 8.1. Experimental Setup

Experiments are conducted on an Intel Pentium PC running Linux (kernel 2.6.x). This PC is equipped with a 2 GHz Intel Mobile processor and 2 GB of RAM. At test time, we stop most unessential background processes. Further, each experimental series is run twice and the shortest obtained time is reported.[23]

[23]Infrequently, some OS daemons, e.g., a swap daemon, would activate and use the processor resources. The probability of such event is greatly decreased, if the series is run twice.

Table III: Natural Language Dataset

| Dataset name | $\lvert\Sigma\rvert$ | String length statistics | | | | |
|---|---|---|---|---|---|---|
| | | 0.2M | 0.4M | 0.8M | 1.6M | 3.2M |
| English synthetic | 27 | 2–27/9.3 | 2–28/9.5 | 2–28/9.6 | 2–28/9.8 | 2–28/10.0 |
| Russian synthetic | 33 | 2–25/10.5 | 2–25/10.5 | 2–25/10.6 | 2–25/10.7 | 2–25/10.9 |
| ClueWeb09 | 36 | 1–32/6.9 | 1–32/7.2 | 1–32/7.4 | 1–32/7.8 | 1–32/8.1 |

*Notes:* String length statistics is given in the form: minimum–maximum/average length. M stands for million dictionary entries.

All search methods were implemented in C++ from scratch.[24] The correctness of most implementations has been checked using automated tests. The exception is our simplified implementation of the $k$-errata tree by Cole et al. [2004], which supports only membership queries and, therefore, has been tested less rigorously.

Automated testing involves building an index over the dictionary $W'$ that contains 5 thousand strings. Then for the first hundred strings $s \in W'$, we generate a full one-neighborhood, a partial two-neighborhood, and a partial three-neighborhood. For each string $s'$ from a generated $k$-neighborhood, the automated test checks whether the original string $s$ can be found in the index over $W'$ within $k$ errors using the string $s'$ as the pattern.

In addition, we carry out preliminary tests to determine the optimal parameters that include a signature size, vector size, size of the reduced alphabet, $q$-gram size, and bucket size for the BKT.

*8.1.1. Performance Measures and Relationships of Interest.* We use the following performance measures:

— Average in-memory retrieval time (all indices are stored in RAM; external memory, i.e., hard disk, is not involved at search time).
— Index size.
— Filtering efficiency, which is calculated as $1 - N_{check}/N$. $N_{check}$ is the number of verifications performed during the checking step. Because some of the methods may verify the same string more than once, the number of verifications can be greater than the number of dictionary strings $N$. In such a case, the filtering efficiency is negative.
— Ratio of retrieval time for a transposition-aware implementation to retrieval time for a transposition-unaware implementation, i.e., the "cost" of treating transposition as a basic edit operation.

Our primary goal is to study relationships between retrieval time and:

— the maximum allowed edit distance $k$;
— the number of dictionary strings $N$;
— the index size;
— the length of the pattern and/or indexed strings.

*8.1.2. Datasets.* We use natural language datasets and DNA datasets. Natural language datasets include: Russian and English synthetic datasets as well as a web dataset ClueWeb09. Each natural language dataset consists of five dictionaries containing 0.2, 0.4, 0.8, 1.6, and 3.2 million unique strings. Their characteristics are summarized in Table III.

[24]We have also implemented a version of inverted $q$-gram file that uses Berkeley DB, but it turned out to be inferior to a custom implementation.

Table IV: DNA Dataset

| Dataset name | $\lvert\Sigma\rvert$ | String length range | Range for the number dictionary strings | Number of dictionaries |
|---|---|---|---|---|
| DNA-11 | 4 | 11 | 0.2M–3.2M | 5 |
| DNA-20 | 4 | 20 | 0.2M–3.2M | 5 |
| DNA-Var | 4 | 11–20 | 0.2M | 10 |

*Note:* M stands for million dictionary entries.

*Synthetic datasets* are generated using a Markov model: the probability of any specific character occurrence in the $i$-th position of string $s$ is a function of $i$, the string length, and of the preceding trigram $s_{[i-3:i-1]}$. The probabilities are estimated using the real English and Russian dictionaries obtained from a publicly available distribution of the open-source spell-checker Ispell [Kuenning et al. 1988]. The alphabet of the synthetic English dataset has 27 characters: 26 Latin letters and an apostrophe. The alphabet of the synthetic Russian dataset consists of 33 Cyrillic characters.

To ensure that the implemented methods produce similar results on real and synthetic data, we create synthetic dictionaries containing the same number of entries as original Ispell dictionaries. Then we compare retrieval times obtained for synthetic dictionaries and Ispell dictionaries.

*The web dataset* consists of frequent words extracted from the subset B of the TREC[25] collection ClueWeb09.[26] This dataset contains mostly English web-pages. During extraction all punctuation is ignored. In addition, we exclude all words with characters other than Latin letters or digits. ClueWeb09 includes five dictionaries containing 0.2, 0.4, 0.8, 1.6, and 3.2 million most frequently occurring words, respectively. Among all our datasets, ClueWeb09 has the largest alphabet of 36 characters: 26 Latin letters and 10 digits.

*DNA datasets* include DNA-11, DNA-20, and DNA-Var. They contain substrings randomly extracted from a human genome.[27] The alphabet of any DNA set has only 4 characters: `A`, `G`, `C`, `T`. During random extraction, a small number of DNA pieces containing a wildcard character `N` are ignored. DNA-11 consists of sequences that are 11 characters long. This is the minimum length $l$ such that there exist 3.2 million unique DNA sequences of the length $l$. DNA-20 contains strings of the length 20. The characteristics of DNA datasets are summarized in Table IV.

Similar to natural language datasets both DNA-11 and DNA-20 include five dictionaries containing 0.2, 0.4, 0.8, 1.6, and 3.2 million DNA sequences. DNA-Var encompasses 10 dictionaries $W_i$ ($11 \leq i \leq 20$). Each $W_i$ contains 0.2 million DNA sequences of the length $i$.

*8.1.3. Testing Approach.* The testing procedure is carried out in two steps. First, we create an in-memory index and evaluate average retrieval time. Next, we build a compact index in external memory and evaluate the size of the index. This step is essentially a serialization of the in-memory index.

There are three different test scenarios:

(1) The first scenario is designed to study how retrieval time depends on the number of dictionary strings $N$ and the maximum allowed edit distance $k$. This scenario

[25] Text REtrieval Conference, http://trec.nist.gov
[26] http://boston.lti.cs.cmu.edu/Data/clueweb09/
[27] http://genome.ucsc.edu/ and http://hgdownload.cse.ucsc.edu/downloads.html

Table V: Summary of implemented methods

| Method type | Method name | Search complexity (upper bound for the average case) | Index overhead |
|---|---|---|---|
| | | **Direct methods** | |
| **Prefix Trees** | String trie | $O\left(n^{k+1}\|\Sigma\|^k\right)$ | $O(N(P+1))$ |
| | FB-trie | N/A | $O(N(P+1))$ |
| | $k$-errata tree | $O\left(n + \frac{(6\log_2 N)^k \log\log \lambda N}{k!} + 3^k \cdot occ\right)$ | $O\left(N \frac{(5\log_2 N)^k}{k!}\right)$ |
| **Neighborhood generation** | Full neighborhood generation | $O\left(n^{k+1}\|\Sigma\|^k\right)$ | $O(N)$ |
| | Super-condensed neighborhood generation | N/A | $O(\lambda N)$ |
| | Reduced alphabet generation | $O\left(n^{k+1}\|\sigma\|^k\left\{1 + \frac{L(n,k)}{\|\sigma\|^n}\right\}\right)$ | $O(M_1(N))$ |
| | Mor-Fraenkel method | $O\left(n^{k+1}\left\{1 + \frac{n^{k-1}L(n,k)}{\|\Sigma\|^{n-k}}\right\}\right)$ | $O((\lambda_m)^{k+1}N)$ |
| | Hybrid neighborhood generation-$t$ | $O\left(n^{k+1}\|\Sigma\|^{k-k'}\left\{1 + \frac{n^{k'-1}L(n,k')}{\|\Sigma\|^{n-k'}}\right\}\right)$, $k' = \min(k,t)$ | $O((\lambda_m)^{t+1}N)$ |
| **Metric space methods** | BKT | $O\left(N^{\beta_1(k)}\right)$, $\beta_1(k) < 1$ | $O\left(\lambda_m P \min\left(N, \left(\frac{N}{b}\right)^\gamma\right)\right)$, for some $\gamma > 0$ |
| | | **Sequence-based filtering methods** | |
| **Pattern partitioning** | Length-divided inverted $q$-gram index | $O(\beta_2(k)N)$ | $O(\lambda_m\|\Sigma\|^q + \alpha_{\text{comp}}\lambda N P)$ |
| **Mapping to vector space** | Unigram frequency vector trie | $O(\beta_3(k)N)$ | $O(mM_2(N))$ |
| | $Q$-gram frequency vector trie | $O(\beta_4(k)N)$ | $O(mM_3(N))$ |
| | Signature hashing | $O(\beta_5(k)N)$ | $O(2^m P)$ |

*Notes:* $N$ is the number of dictionary strings, $n$ is the pattern size, $q$ is the size of the $q$-gram;
$\lambda$ is the average length of the dictionary string, $\lambda_m$ is the maximum length of the dictionary string;
$L(n,k) = \max_{|i-n|\le k} N(i)$, where $N(i)$ is the number of dictionary strings containing $i$ characters;
$occ$ is the number of approximate pattern occurrences in the dictionary;
$|\Sigma|$ denotes the size of the original alphabet, $|\sigma|$ denotes the size of the reduced alphabet;
$m$ is the size of the frequency vector or the signature;
$M_i(N)$ is the number of ($q$-gram) frequency vectors or reduced dictionary strings, usually $M_i(N) \ll N$;
$P$ is the pointer size, $\alpha_{\text{comp}}$ is the compression ratio, $\beta_i(k)$ are positive-valued functions.

Table VI: Method Parameters

| | Synthetic data sets (English and Russian) | | | ClueWeb09 | | | DNA data sets | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\|\sigma\|$ | $q$-gram size | bucket size | $\|\sigma\|$ | $q$-gram size | bucket size | $\|\sigma\|$ | $q$-gram size | bucket size |
| BKT | | | 200 | | | 200 | | | 75 |
| Length-divided inverted $q$-gram file | | 4 | | | 3 | | | 4 | |
| $q$-gram frequency vector trie | 9 | 2 | | 9 | 2 | | 8 | 2 | |
| Reduced alphabet neighborhood generation | 3 | | | 5 | | | 3 | | |
| Signature hashing | 13 | | | 13 | | | | | |
| Unigram frequency vector trie | 10 | | | 10 | | | 4 | | |

*Note:* $|\sigma|$ is the size of the reduced alphabet, the frequency vector, or the signature.

uses synthetic datasets (English and Russian), ClueWeb09, DNA-11, and DNA-20. Each dataset is represented by five dictionaries containing from 0.2 to 3.2 million strings. However, due to memory restrictions, some of the methods are tested using only a subset of smaller size dictionaries.
For every dictionary and edit distance $k = 1, 2, 3$, we create a set of 1000 test patterns in two stages. First, we randomly select a dictionary string. Second, we randomly select the pattern from a $k$-neighborhood of the selected string.[28]

(2) The second scenario is designed to study how retrieval time depends on the length of indexed strings. It employs the dataset DNA-Var, which includes 10 dictionaries. Each dictionary contains 0.2 million DNA sequences of the same length (see Table IV). The patterns are generated from dictionary strings by applying up to $k$ edit operations similarly to the first scenario.

(3) In the third scenario, which is more appropriate for natural languages, there is a single dictionary and multiple sets of patterns. This scenario is designed to study the relationship between retrieval time and pattern length. The dictionary contains 0.2 million strings (of various lengths). Each set of patterns contains 1000 strings of the same length (generated similarly to the first scenario). There are 11 sets of patterns of lengths varying from 5 to 15. We use this scenario with two datasets: synthetic English and ClueWeb09.

### 8.2. Implemented Methods and Comparison Baseline

The implemented search methods are listed in Table V. In columns three and four we summarize well-known theoretical results for the upper bound of average retrieval time and index overhead, i.e., the arithmetic difference between the size of the index and the size of the dictionary. Details are provided in Appendix C. Empirically determined parameters of the methods are listed in Table VI. In the following subsections, we briefly review implemented methods. We also describe a sequential search method used as a comparison baseline.

*8.2.1. Direct Indexing Methods.* Direct methods are represented by prefix trees, neighborhood generation, and metric space methods.

We have implemented the following modifications of *prefix trees*:

— The string trie is a tree where strings with a common prefix belong to the same subtree. The search algorithm is essentially a parallel traversal of the tree and the Levenshtein automaton (Section 6.1.1, p. 20).

— The FB-trie uses an additional trie built over reversed dictionary strings to improve performance over the regular string trie (Section 6.1.2, p. 22). FB-trie is a hybrid method that combines prefix trees and pattern partitioning.

— The $k$-errata tree blends partial neighborhood generation with tries and treats errors by recursively creating errata trees from subtrees of the regular trie (Section 6.1.3, p. 23). This process involves merging of subtrees: to reduce memory requirements, we use shallow copying whenever possible. A longest common prefix index proposed as a part of $k$-errata tree does not make much sense for short patterns. Therefore, we do not use it in our implementation. Note that our simplified implementation fully supports only membership queries.

Implemented *neighborhood generation* methods include the classic algorithm as well as several hybrid methods:

[28]In the case of $k = 1$, we choose the pattern from the full $k$-neighborhood, for $k = 2$ and $k = 3$, however, we use a partial neighborhood. For $k = 3$ and natural language data, most neighborhood generation methods are slow. Therefore, only 100 patterns are used.

— Full neighborhood generation is a well-known folklore method that computes the full $k$-neighborhood of the pattern (all strings within distance $k$) and searches for its members for an exact match using hashing (Section 6.2, p. 25).

— Super-condensed neighborhood generation is a filtering method that employs a substring-free, i.e., super-condensed, neighborhood to retrieve candidate strings (Section 6.2.3, p. 27). Substring dictionary searching is carried out using an inverted file, which indexes substrings obtained from a dictionary string by deleting $i_1$ prefix characters and $i_2$ suffix characters (for all $i_1 + i_2 \leq 2k$). This implementation is a proof of concept: the inverted file is fast, but not space-efficient. Because generation of a super-condensed neighborhood is slow in the case of a large alphabets and natural language data, we test it using only DNA data.

— Reduced-alphabet neighborhood generation is a filtering method that employs a lossy string representation obtained by alphabet hashing (Section 6.2.5, p. 28). The reduced alphabet is small and it takes little time to generate the full neighborhood. The checking step of the method is very efficient: it has linear time with respect to the total length of candidate strings.

— The Mor-Fraenkel method is an intermediate approach that computes deletion-only $l$-neighborhoods ($l \leq k$) of dictionary strings during the indexing. At query time, it computes the deletion-only $l$-neighborhoods of the pattern and searches for its members exactly. The case $k = 1$ is explained in Section 6.2.7, p. 30; the general case is presented in Section 6.2.8, p. 30 and in Section 6.2.9, p. 31.

— Hybrid neighborhood generation methods combine full query-time neighborhood generation with Mor-Fraenkel method (Section 6.2.11, p. 34). The rationale behind this approach is to reduce space requirements of Mor-Fraenkel method, which is often impractical for $k > 1$. We have implemented two modifications of hybrid neighborhood generation. Hybrid neighborhood generation-1 uses less memory and precomputes one-deletion neighborhoods. Hybrid neighborhood generation-2 precomputes both one-deletion and two-deletion neighborhoods. For short patterns, we resort to full neighborhood generation: as a rule of thumb, full neighborhood generation is applied only when $k = 3$ and $|p| < 5$.

Implemented *metric space* methods include BKTs and its modifications:

— The BKT recursively divides the space based on the distances to pivot elements (Section 6.3, p. 34). The process stops when the number of strings in each partition (i.e., a bucket) becomes less than or equal to $b$ (bucket size). At query time, the BKT prunes the space using the triangle inequality. This requires calculating distances from the pattern to selected pivots. Note that strings inside the buckets are compared with the pattern using a fast online checking algorithm outlined in Section 5, p. 19.

— The multi-pivot modification of the BKT that uses more than one pivot per node.

— The hybrid of the BKT and the AESA (Section 6.3.4, p. 36). At the top level this method behaves as the BKT. Strings inside each bucket are indexed using the AESA: for each pair of strings $s_i$ and $s_j$ in the bucket, the method precomputes and stores the edit distance $\text{ED}(s_i, s_j)$ between them.

In the case of DNA dictionaries, the distance to pivots is computed using an efficient bit-parallel algorithm proposed by Myers [1999]. This algorithm was extended by Hyyrö [2005] to handle transpositions. Note, however, that the Myers-Hyyrö algorithm computes the so-called *restricted* Damerau-Levenshtein distance, which does not always satisfy the triangle inequality (see Section 2.2 for a detailed discussion). As a result, the respective transposition-aware implementation is prone to a small number of false misses, which accounts for 0.1 to 0.2 percent of failures during our functional testing.

Given our focus on lossless solutions that guarantee retrieval of strings within a specified edit distance $k$, we do not use this algorithm for natural language dictionaries. Instead, we employ a less efficient algorithm of Lowrance and Wagner [1975] (see Equation (6) on p. 11 and the pseudo-code in Appendix D.3). This does lead to performance deterioration, but the difference is not significant.

*8.2.2. Indexing Methods Based on Sequence Filtering.* The methods in these groups include pattern partitioning methods and vector space frequency distance methods.

*Pattern partitioning* methods are represented by:

— The *length-divided $q$-gram file* (Section 7.1.2, p. 38). The idea of the method consists in creating an inverted file that allows one to find all strings containing a given $q$-gram. The inverted lists are sorted by respective string identifiers and compressed using the variable-byte encoding (see Appendix C.5.3). The length-divided $q$-gram file can be considered as a hybrid method, because it blends pattern partitioning with division of the index based on the string length. For patterns shorter than $k + 1$, it uses sequential searching on the subset of dictionary strings shorter than $2k + 1$.

There are several pattern partitioning methods that improve over classic $q$-gram indices: intermediate pattern partitioning [Navarro and Baeza-Yates 2000], indexing of gapped $q$-grams [Burkhardt and Kärkkäinen 2002], $q$-samples [Navarro et al. 2005], and equi-probable multi-grams [Schek 1978; Navarro and Salmela 2009]. Retrieval time can also be improved through computing an optimal partitioning [Navarro and Baeza-Yates 1998]. We have not explored these methods, primarily because they improve over exact pattern partitioning (based on $q$-gram indices) by a small factor, which is usually within an order of magnitude. However, the length-divided $q$-gram file is up to two orders of magnitude slower than the best methods: the FB-trie and the Mor-Fraenkel method (see Table VII).

It is also noteworthy that the dictionary of modern retrieval systems and spell-checkers can be very large, partly because they treat some sequences of adjacent space-separated words as single strings. For example, the spell-checker of the medical search engine PubMed includes more than 14 million unique sequences (containing from one to three words). The number of single words alone can also be large. Even though we use only 3.2 million strings from the subset B of the ClueWeb09 collection, the complete set contains more than 100 million unique strings of which more than 20 million appear at least 3 times in the collection.

However, the retrieval time of filtering methods, which rely on indexing of small string fragments, would linearly depend on the number of dictionary strings. Unlike the fastest *sublinear* methods (FB-tries and the Mor-Fraenkel method), their performance would not scale up well.

*Vector space frequency distance* methods use conversion of strings into unigram and $q$-gram frequency vectors or signatures. Then frequency vectors are indexed using the trie. Original dictionary queries are substituted with frequency distance queries. Vector space methods can be seen as variants of pattern partitioning methods.

We have implemented the following vector space methods:

— The unigram frequency vector trie (Section 7.2.3, p. 45);
— The $q$-gram ($q > 1$) frequency vector trie (Section 7.2.4, p. 45).
— Signature hashing converts strings to short unigram signatures and searches for them sequentially (Section 7.2.5, p. 45). Note that signature hashing is not efficient in the case of small alphabets. Thus, we do not use it for DNA datasets. Unlike unigram and $q$-gram frequency vector tries, signature hashing appears to be sensitive to the choice of the hash function $h(c)$ that is used to reduce dimensionality. Therefore, we

Fig. 7: Efficiency of on-line searching methods, on Synthetic English data (800,000 entries). The panel on the left represents transposition-unaware implementations, the panel on the right represents transposition-aware implementations.

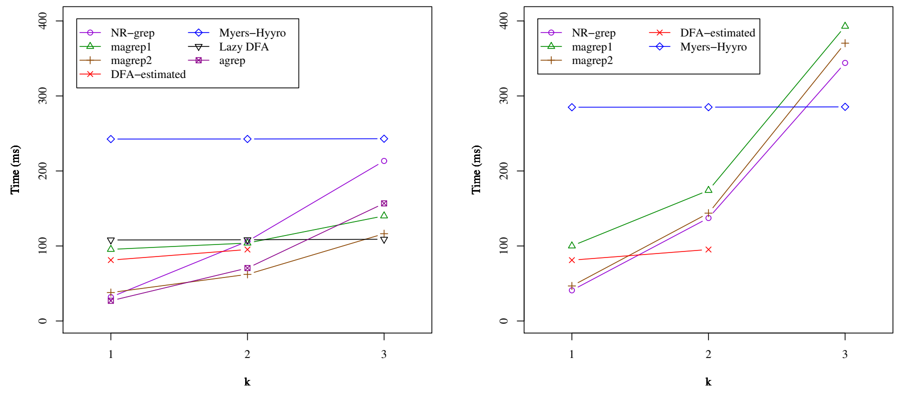
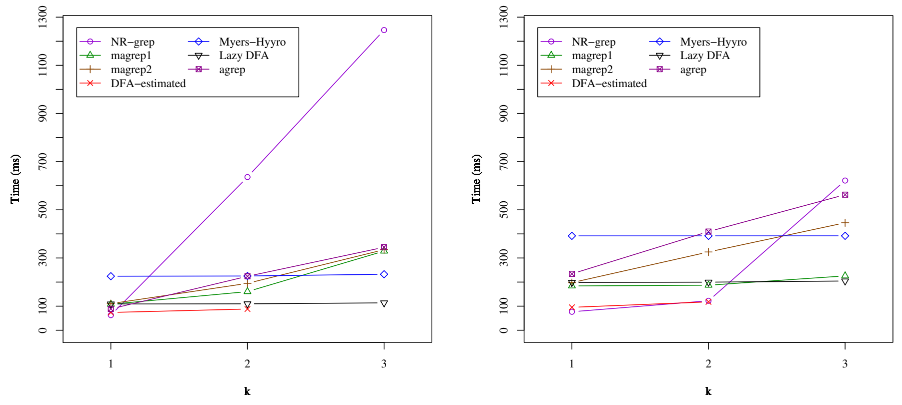


Fig. 8: Efficiency of on-line searching methods, on DNA data (800,000 entries). Transposition-unaware searching. The panel on the left represents a dictionary where sequences are 11 character long, the panel on the right represents a dictionary where sequences are 20 characters long.



use a frequency optimized hash function, which allows one to achieve significantly shorter retrieval time (see Appendix B, p. 78).

*8.2.3. Comparison Baseline.* Retrieval times would be of little value in the absence of a comparison baseline. To this end, we use a combination of two sequential search algorithms: magrep1 and magrep2, which are outlined in Section 5, p. 19. To confirm that the baseline is efficient, we compare our sequential search implementation with the following sequential search methods:

— *NR-grep* [Navarro and Raffinot 2000; Navarro 2001b];

— *agrep* [Wu and Manber 1992a; 1992b] (used only for transposition-unaware searching);[29]
— *Lazy DFA*: the lazy deterministic Levenshtein automaton [Kurtz 1996; Navarro 1997b] (used only for-transposition-unaware searching);[30]
— *Myers-Hyyro*: the bit-parallel algorithm to compute the Damerau-Levenshtein distance, proposed by Myers [1999] and extended by Hyyrö [2005] to handle transpositions;
— *DFA-estimated*: a trie-based implementation of a deterministic Levenshtein automaton (used for $k \leq 2$; construction time is not accounted for).

The deterministic Levenshtein automaton (DFA-estimated) is implemented as a complete trie (i.e., it is not compressed), which stores a complete $k$-neighborhood. Therefore, it is very efficient during searching. Construction of the automaton requires considerable time, which is not included in total running time that we report. The rationale of this method is to estimate a lower bound for a sequential search in the dictionary.

All methods are implemented in the form of applications that read data from a file. We read the file several times before testing, in order for the operating system to cache file contents in the memory. Therefore, the overhead involved in reading the data from the file is small and is identical for every application.

We use one synthetic English dictionary and two DNA dictionaries that contain sequences 11 and 20 characters long. Each dictionary contains 0.8 million strings. In the case of DNA data, only transposition-unaware modifications are tested, but in the case of synthetic English data, we also test transposition-aware modifications.

The results of the experimental comparison are presented in Figure 7 (synthetic English data) and in Figure 8 (DNA data).

Our implementation of sequential searching combines magrep2 for $k = 1$ and magrep1 for $k \geq 2$. Although it is somewhat slower than the best methods, the running times are competitive.

*8.2.4. Associativity Considerations.* We report sizes of indices that do not include any *associated data*, such as string identifiers (also known as satellite data). Associated data is domain specific and varies in size. Nonetheless, with exception of the Mor-Fraenkel method and the $k$-errata tree, pointers to associated data can be embedded into the index virtually without affecting search performance and index size. To separate text strings and pointers one can use a special separator character, e.g., the character with zero ASCII code.

For Mor-Fraenkel method embedding pointers to associated data may be space inefficient. One to solution to this problem is discussed on p. 33. In our simplified implementation of the $k$-errata tree, most search paths follow labels with wildcards, which may represent more than one string. Consequently, our implementation of the $k$-errata tree fully supports only membership queries. As we explain in Section 6.1.3, p. 25, it is possible to attach a list of string identifiers to each node of the $k$-errata tree during the indexing time. However, this approach is less space efficient.

[29]Because agrep is slow for matching complete strings (option -x), we run two instances of agrep connected via a pipe. The first instance works as a filter: it finds all strings that contain a substring approximately matching the pattern. The second instance runs with option -x. It processes the output of the first instance and reports only complete string matches.
[30]We use the implementation by G. Navarro modified for complete-string matching.

## 8.3. Results

*8.3.1. General Observations.* Retrieval time statistics are presented in Table VII and Figures 10-11. Note that the plots include all the available measurements, but the tables report measurements only for dictionaries containing 0.2, 0.8, and 3.2 million strings. There are several bird's-eye view observations:

— The retrieval time of all implemented methods grows exponentially with the maximum allowed edit distance $k$.

— Even though there are many efficient algorithms for $k \leq 2$, retrieval time of most methods for $k = 3$ is either somewhat better (by about an order of magnitude) or worse than that of sequential searching.

— When comparing efficiency between synthetic Russian and English datasets, the methods have very similar retrieval times. Non-neighborhood generation methods mostly "favor" the Russian dataset, which has a larger alphabet and longer strings, but the difference is less than two-fold.

— Apart from two synthetic natural language datasets, there are considerable differences in retrieval times across datasets. In particular, there are two "difficult" data sets: ClueWeb09 and DNA-11, where all non-neighborhood methods exhibit longer retrieval times. In the case of ClueWeb09, neighborhood generation methods also have some of the worst search times, because the alphabet is large.

*8.3.2. Relationship between Retrieval Time, $N$, and $k$.* Average retrieval times for all datasets except DNA-Var are presented in Table VII, p. 56. Figure 9, p. 57, shows the relationship between retrieval time and index size for indices built over the synthetic English and ClueWeb09 datasets. Because the synthetic Russian dataset yields results almost identical to those of the synthetic English dataset, we do not plot the Russian dataset results. Figure 10, p. 58, presents results for DNA data. Each search method is represented by a polygonal curve, where vertices correspond to dictionaries of different sizes.

We divide all methods into four groups based on their performance. The first group includes prefix trees: the string trie, the FB-trie, and the $k$-errata tree. The second group includes neighborhood generation methods: full, super-condensed, and reduced alphabet neighborhood generation, the Mor-Fraenkel method and hybrid neighborhood generation methods.

Depending on the dataset, $k$, and the amount of available memory, the best retrieval times in our experimental setup are obtained using a method from either the first or the second group: the FB-trie, the Mor-Fraenkel method, or a hybrid neighborhood generation method.

For DNA-20, as well for synthetic Russian and English datasets, the best average retrieval time is within 3 milliseconds. It can also be seen that for these datasets and methods of the first and the second group, the retrieval time virtually does not depend on the number of dictionary strings. If $k$ is sufficiently small, these methods have strictly sublinear retrieval time. For difficult datasets DNA-11 and ClueWeb09, the best average retrieval time is within 30 milliseconds.

Retrieval time of full neighborhood generation is $O\left(n^{k+1}|\Sigma|^k\right)$. In the case of DNA data, the alphabet contains only four characters and full neighborhood generation is one of the fastest methods even for $k = 3$. Super-condensed neighborhood generation is almost always slower than full neighborhood generation, and marginally outperforms the latter method only for DNA-11 and $k = 3$. Also note that retrieval time of full neighborhood generation is nearly constant across dictionaries with different number of strings.

In the case of natural language data, the size of the alphabet is large and retrieval time grows rapidly with $k$. As a result, full neighborhood generation is the slowest method for $k = 3$. It is possible to combine query-time full neighborhood generation with index-time generation of deletion-only neighborhoods. The resulting methods, which we call hybrid neighborhood generation methods, outperform the full neighborhood generation by up to two orders of magnitude for $k = 1$ and $k = 2$. Unfortunately, it is not a space-efficient solution and it does not worked very well for $k = 3$. A better solution is reduced alphabet generation, which works well even for $k = 3$. It outperforms the string trie in most cases.

It is also noteworthy that the regular string trie is not especially efficient for the difficult dataset ClueWeb09. We suppose that this happens because ClueWeb09 is prefix-dense, i.e., it contains a lot of short strings. Consequently, the search algorithm cannot backtrack early and visits more trie nodes in the case of prefix-dense datasets. For the string trie built over the largest ClueWeb09 dictionary, the average number of children for the first four levels are 36, 36, 34.7, and 10.2. In comparison, in the case of the trie built over the largest synthetic English dictionary, the average number of children for the first three levels are 26, 18.5, 9.2, and 4.7.

To verify whether short strings affect performance of the string trie, we measured retrieval time for a dictionary that did not contain strings shorter than 5 characters. Even though the number of such strings is less than 9 percent of the total number of ClueWeb09 strings, their exclusion leads to a 1.6-fold improvement in retrieval time for $k = 3$. Removal of strings that are shorter than 6 characters, which represent only one fifth of all dictionary strings, leads to a 2-fold improvement for $k = 3$.

Retrieval time of the $k$-errata tree is similar to that of the FB-trie: the $k$-errata tree is faster than the FB-trie for the ClueWeb09 dataset, but about 2-3 times slower for other datasets. The method is reasonably fast for $k = 1$, but has a huge index: it is also the only method that could not be tested for $k > 1$ due to memory restrictions.

DNA-11 is also a difficult dataset, which has dense dictionaries. In the case of a dense dictionary it is hard to outperform full neighborhood generation, because such dictionary contains a significant fraction of strings from pattern neighborhoods. In the case of DNA-11 and $k \leq 2$, the only methods outperforming full neighborhood generation are Mor-Fraenkel method and hybrid neighborhood generation methods. However, both neighborhood generation methods are inefficient for $k = 3$.

In addition, the gap in performance between full neighborhood generation and the other methods rapidly decreases as $N$ increases. For the smallest dictionary and $k = 1$, hybrid neighborhood generation-1 is almost three times faster than full neighborhood generation. Yet, for $N = 1.6 \times 10^6$, where the DNA-11 dictionary contains 38 percent of all 11-character DNA-sequences, the difference between hybrid neighborhood generation-1 and full neighborhood generation is only 1.5-fold. An analytical justification of this observation is given in Section C.3.5, p. 84. For $k = 3$, FB-trie can be up to 16 times faster, but also only for small $N$. For $N = 3.2 \times 10^6$, FB-trie and full neighborhood generation have equivalent performance.

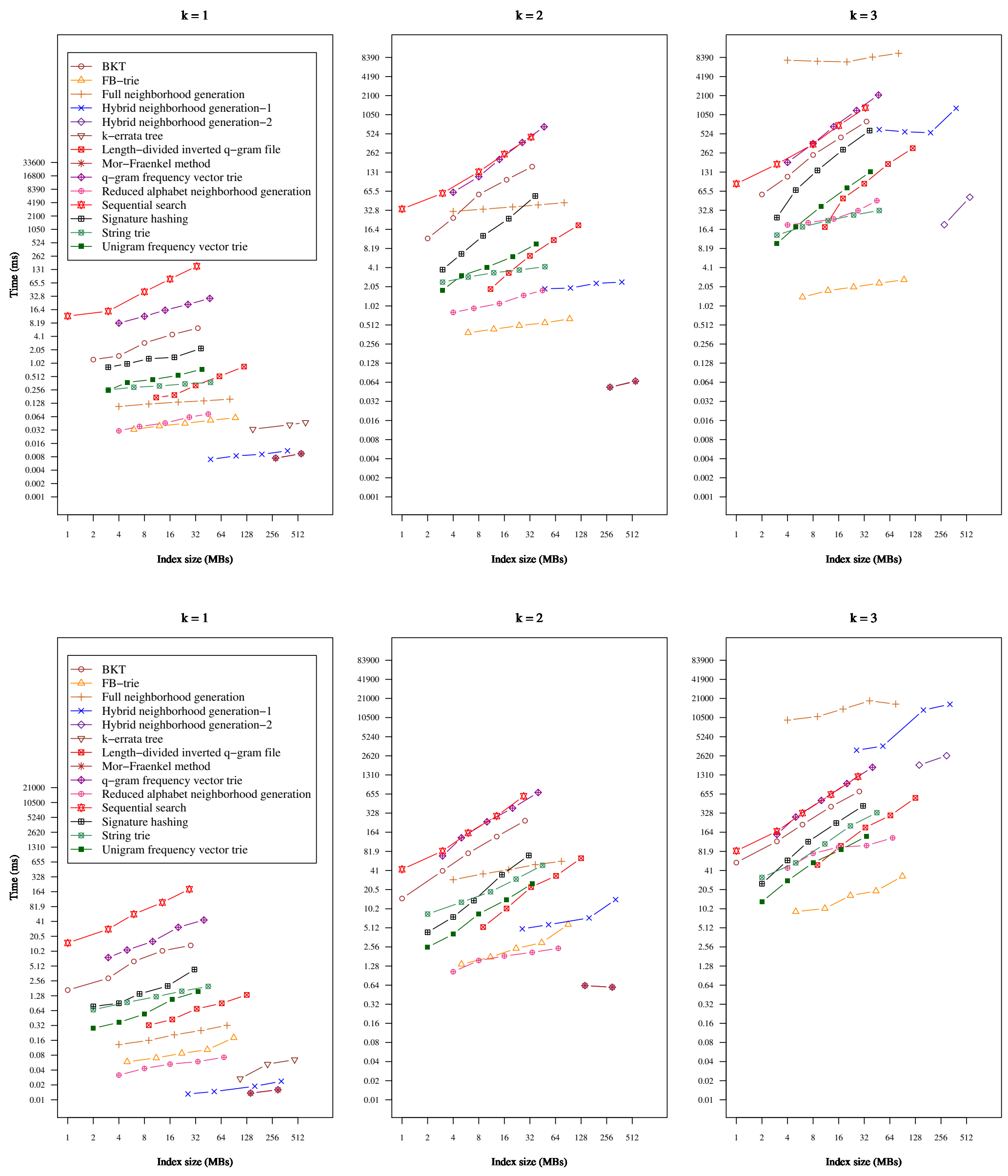
The methods in the third group include a unigram frequency vector trie, signature hashing, and the length-divided $q$-gram file. They are 10 to 100 times slower than the methods in the first group, but are still 10 to 100 times faster than sequential searching for $k \leq 2$. Retrieval time of these methods linearly depends on $N$ and increases faster than that of methods from the first group. For $k = 3$ and $N = 3.2 \times 10^6$, only unigram

Table VII: In-memory search time (ms)

| | $k=1$ | | | $k=2$ | | | $k=3$ | | |
|---|---|---|---|---|---|---|---|---|---|
| **Number of dictionary strings** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** |
| **English synthetic** | | | | | | | | | |
| BKT | 1.230 | 2.913 | 6.278 | 11.79 | 58.16 | 159.4 | 58.12 | 243.3 | 820.0 |
| FB-trie | 0.033 | 0.045 | 0.060 | 0.386 | 0.499 | 0.637 | 1.405 | 2.019 | 2.632 |
| Full neighborhood generation | 0.108 | 0.136 | 0.158 | 31.38 | 36.95 | 43.12 | 7574 | 7131 | 9711 |
| Hybrid neighborhood generation-1 | 0.007 | 0.009 | | 1.896 | 2.314 | | 611.4 | 546.5 | |
| Hybrid neighborhood generation-2 | 0.007 | | | 0.053 | | | 19.41 | | |
| $k$-errata tree | 0.033 | 0.047 | | | | | | | |
| Length-divided inverted $q$-gram file | 0.172 | 0.323 | 0.855 | 1.879 | 6.277 | 19.02 | 17.83 | 86.05 | 311.1 |
| Mor-Fraenkel method | 0.007 | | | 0.053 | | | | | |
| $q$-gram frequency vector trie | 8.164 | 15.93 | 29.34 | 62.85 | 207.3 | 676.7 | 186.4 | 679.8 | 2142 |
| Reduced alphabet neighborhood generation | 0.030 | 0.046 | 0.073 | 0.806 | 1.111 | 1.787 | 19.24 | 23.86 | 46.41 |
| **Sequential search** | **11.74** | **41.26** | **155.2** | **34.31** | **132.2** | **466.1** | **85.60** | **358.7** | **1350** |
| Signature hashing | 0.830 | 1.287 | 2.193 | 3.803 | 12.96 | 54.96 | 25.14 | 139.0 | 589.8 |
| String trie | 0.256 | 0.313 | 0.377 | 2.417 | 3.401 | 4.236 | 13.30 | 22.47 | 32.48 |
| Unigram frequency vector trie | 0.248 | 0.438 | 0.749 | 1.809 | 4.162 | 9.640 | 9.756 | 37.81 | 132.1 |
| **Russian synthetic** | | | | | | | | | |
| BKT | 0.812 | 2.386 | 5.107 | 9.320 | 45.66 | 124.0 | 50.49 | 203.4 | 626.8 |
| FB-trie | 0.029 | 0.041 | 0.082 | 0.286 | 0.381 | 0.709 | 0.874 | 1.312 | 2.059 |
| Full neighborhood generation | 0.147 | 0.174 | 0.201 | 57.70 | 66.51 | 75.14 | 19987 | 17704 | 21130 |
| Hybrid neighborhood generation-1 | 0.007 | 0.008 | | 2.741 | 2.991 | | 1547 | 1408 | |
| Hybrid neighborhood generation-2 | 0.007 | | | 0.046 | | | 26.47 | | |
| $k$-errata tree | 0.033 | 0.046 | | | | | | | |
| Length-divided inverted $q$-gram file | 0.105 | 0.309 | 0.945 | 0.912 | 3.510 | 11.57 | 9.312 | 46.31 | 185.3 |
| Mor-Fraenkel method | 0.007 | | | 0.046 | | | | | |
| $q$-gram frequency vector trie | 7.957 | 16.94 | 31.17 | 61.39 | 177.7 | 543.8 | 196.4 | 654.3 | 2050 |
| Reduced alphabet neighborhood generation | 0.027 | 0.043 | 0.068 | 0.778 | 1.137 | 1.759 | 21.04 | 28.71 | 41.84 |
| **Sequential search** | **5.970** | **35.54** | **139.9** | **19.67** | **97.12** | **352.0** | **70.88** | **281.9** | **1064** |
| Signature hashing | 0.761 | 1.155 | 2.068 | 3.172 | 11.75 | 55.90 | 30.52 | 122.6 | 477.4 |
| String trie | 0.232 | 0.274 | 0.469 | 2.013 | 2.820 | 3.701 | 11.63 | 19.49 | 28.64 |
| Unigram frequency vector trie | 0.359 | 0.488 | 0.561 | 2.086 | 3.645 | 7.313 | 8.668 | 31.44 | 80.72 |
| **ClueWeb09** | | | | | | | | | |
| BKT | 1.667 | 6.344 | 13.39 | 14.81 | 76.53 | 248.8 | 54.53 | 216.2 | 718.7 |
| FB-trie | 0.058 | 0.087 | 0.180 | 1.366 | 2.420 | 5.763 | 9.200 | 16.48 | 33.17 |
| Full neighborhood generation | 0.131 | 0.207 | 0.321 | 29.31 | 42.03 | 56.93 | 9587 | 14300 | 17117 |
| Hybrid neighborhood generation-1 | 0.013 | 0.018 | | 4.926 | 7.326 | | 3211 | 13775 | |
| Hybrid neighborhood generation-2 | 0.013 | | | 0.629 | | | 1874 | | |
| $k$-errata tree | 0.026 | 0.065 | | | | | | | |
| Length-divided inverted $q$-gram file | 0.324 | 0.696 | 1.332 | 5.255 | 22.39 | 63.81 | 49.93 | 194.5 | 569.7 |
| Mor-Fraenkel method | 0.013 | | | 0.629 | | | | | |
| $q$-gram frequency vector trie | 7.604 | 16.08 | 43.75 | 69.41 | 239.2 | 693.6 | 150.6 | 519.2 | 1728 |
| Reduced alphabet neighborhood generation | 0.031 | 0.052 | 0.072 | 1.040 | 1.839 | 2.436 | 44.77 | 95.50 | 133.5 |
| **Sequential search** | **15.03** | **57.47** | **183.9** | **42.74** | **159.8** | **606.7** | **83.38** | **328.4** | **1233** |
| Signature hashing | 0.778 | 1.395 | 4.352 | 4.354 | 13.73 | 70.68 | 25.30 | 116.4 | 426.0 |
| String trie | 0.666 | 1.227 | 1.979 | 8.435 | 18.89 | 49.37 | 31.80 | 107.3 | 333.3 |
| Unigram frequency vector trie | 0.280 | 0.545 | 1.542 | 2.532 | 8.490 | 25.33 | 13.18 | 54.33 | 141.0 |
| **DNA-11** | | | | | | | | | |
| BKT | 1.829 | 4.492 | 11.85 | 14.34 | 45.47 | 153.5 | 36.36 | 127.3 | 467.4 |
| FB-trie | 0.049 | 0.088 | 0.226 | 0.716 | 1.535 | 4.504 | 2.690 | 8.062 | 31.22 |
| Full neighborhood generation | 0.022 | 0.027 | 0.033 | 0.912 | 1.085 | 1.259 | 31.26 | 36.84 | 40.78 |
| Hybrid neighborhood generation-1 | 0.008 | 0.013 | | 0.695 | 6.275 | | 451.9 | 12646 | |
| Hybrid neighborhood generation-2 | 0.008 | | | 0.091 | | | 133.6 | | |
| $k$-errata tree | 0.055 | 0.080 | | | | | | | |
| Length-divided inverted $q$-gram file | 1.191 | 5.068 | 27.54 | 11.33 | 66.41 | 254.2 | 61.41 | 368.7 | 1430 |
| Mor-Fraenkel method | 0.008 | | | 0.091 | | | | | |
| $q$-gram frequency vector trie | 1.551 | 4.579 | 16.78 | 15.43 | 55.19 | 205.4 | 66.02 | 246.2 | 942.0 |
| Reduced alphabet neighborhood generation | 0.031 | 0.058 | 0.092 | 0.724 | 1.466 | 2.275 | 16.36 | 30.39 | 75.77 |
| **Sequential search** | **26.54** | **106.8** | **415.4** | **46.91** | **187.8** | **741.2** | **80.35** | **323.6** | **1259** |
| String trie | 0.212 | 0.217 | 0.442 | 2.606 | 3.201 | 7.570 | 16.25 | 32.03 | 67.87 |
| Super-condensed neighborhood generation | 0.082 | 0.092 | 0.125 | 1.475 | 1.706 | 2.832 | 17.47 | 21.61 | 52.15 |
| Unigram frequency vector trie | 1.652 | 6.873 | 29.88 | 7.690 | 31.88 | 138.2 | 31.54 | 133.6 | 545.0 |
| **DNA-20** | | | | | | | | | |
| BKT | 1.479 | 3.460 | 8.024 | 8.494 | 27.18 | 82.36 | 44.17 | 153.2 | 529.0 |
| FB-trie | 0.016 | 0.019 | 0.037 | 0.175 | 0.271 | 0.653 | 0.341 | 0.528 | 1.364 |
| Full neighborhood generation | 0.040 | 0.046 | 0.050 | 3.046 | 3.465 | 3.714 | 205.8 | 205.9 | 229.0 |
| Hybrid neighborhood generation-1 | 0.010 | 0.011 | | 1.047 | 1.178 | | 87.32 | 98.25 | |
| $k$-errata tree | 0.057 | | | | | | | | |
| Length-divided inverted $q$-gram file | 1.354 | 7.493 | 39.46 | 3.221 | 19.47 | 85.69 | 9.737 | 50.31 | 210.4 |
| $q$-gram frequency vector trie | 1.708 | 3.314 | 6.606 | 11.17 | 25.96 | 62.10 | 40.67 | 107.8 | 306.0 |
| Reduced alphabet neighborhood generation | 0.040 | 0.044 | 0.054 | 2.083 | 2.277 | 2.453 | 113.5 | 125.3 | 134.9 |
| **Sequential search** | **46.55** | **186.3** | **740.6** | **77.29** | **310.1** | **1238** | **108.0** | **431.7** | **1725** |
| String trie | 0.234 | 0.228 | 0.452 | 2.901 | 3.344 | 7.519 | 18.22 | 34.81 | 63.53 |
| Super-condensed neighborhood generation | 0.225 | | | 11.24 | | | 416.7 | | |
| Unigram frequency vector trie | 0.935 | 3.518 | 13.30 | 2.387 | 9.712 | 36.21 | 8.230 | 30.80 | 122.5 |

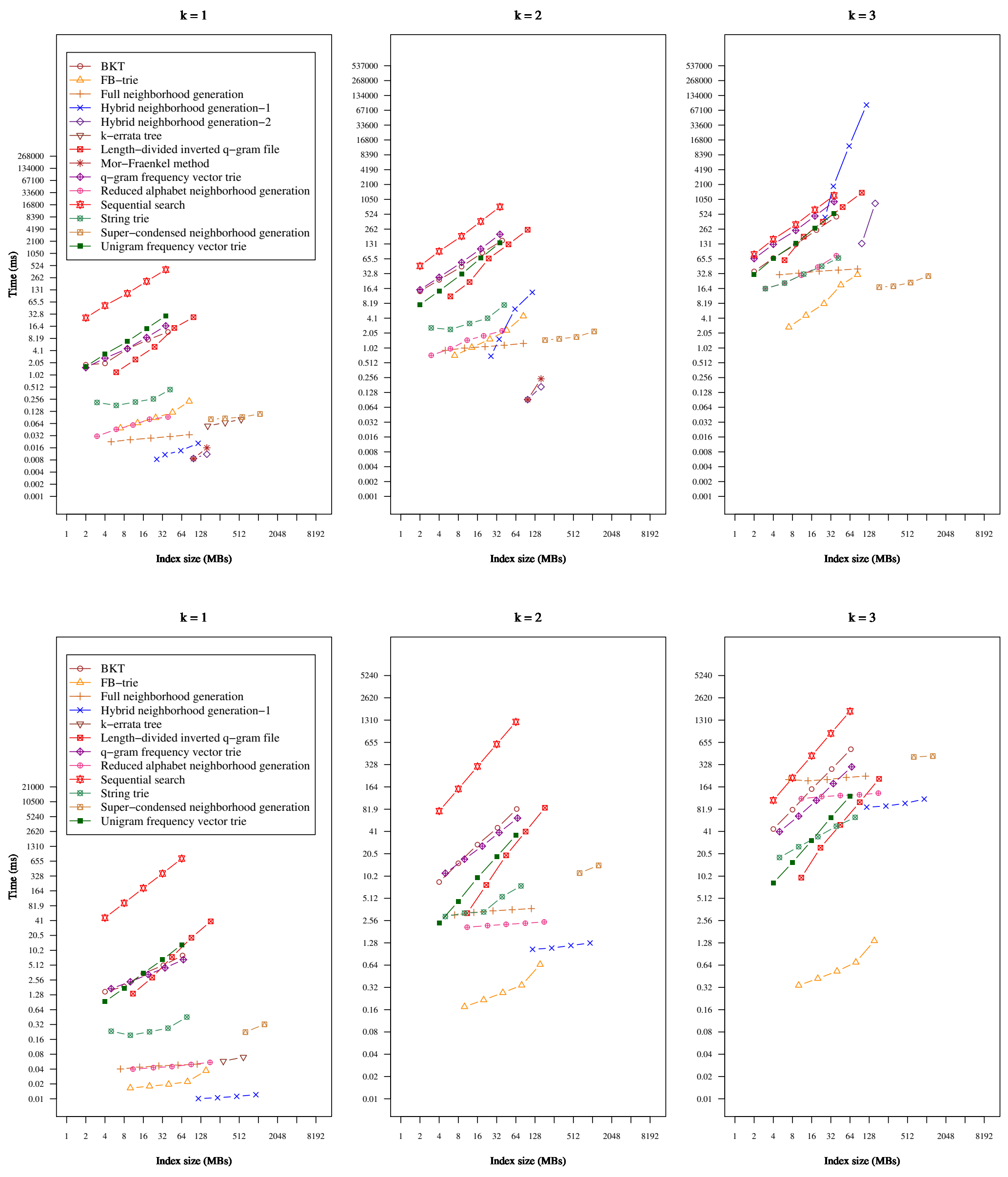
*Note:* M stands for million entries.

Fig. 9: The relationship between retrieval time and index size (natural language data). The first row represents the synthetic English dataset, the second row represents the ClueWeb09 dataset.



*Notes:* Logarithmic scale on both axes. Each search method is represented by a polygonal curve with vertices that correspond to dictionaries of sizes 0.2, 0.4, 0.8, 1.6, and 3.2 million strings. Most methods are represented by five test dictionaries. Hybrid neighborhood generation-1, Mor-Fraenkel method, hybrid neighborhood generation-2, super-condensed neighborhood generation, and the $k$-errata tree are tested using fewer dictionaries due to memory restrictions.

Fig. 10: The relationship between retrieval time and index size (DNA data). The first row represents DNA-11, the second row represents DNA-20.



*Notes:* Logarithmic scale on both axes. Each search method is represented by a polygonal curve with vertices that correspond to dictionaries of sizes 0.2, 0.4, 0.8, 1.6, and 3.2 million strings. Most methods are represented by five test dictionaries. Hybrid neighborhood generation-1, Mor-Fraenkel method, hybrid neighborhood generation-2, super-condensed neighborhood generation, and the $k$-errata tree are tested using fewer dictionaries due to memory restrictions.

frequency vector trie can be an order of magnitude faster than sequential searching. Note that neither the unigram frequency vector trie nor the length-divided $q$-gram file are efficient for DNA-11. In this case, both methods are either marginally better or worse than the BKT.

The fourth group includes the BKT and $q$-gram frequency vector tries. In the case of natural language data, these methods are slow. The $q$-gram frequency vector trie is slower than sequential searching for $k = 2, 3$ and $N = 3.2 \times 10^6$. The average retrieval time of the BKT is always better than that of sequential searching, but a significant speedup is achieved only for $k = 1$ and synthetic datasets (Russian and English). For the real-world ClueWeb09 dataset, the maximum improvement over sequential searching (achieved only for $k = 1$) is 14-fold.

In the case of DNA data, both methods have better retrieval times, which are typically 2-3 times shorter than those for natural language datasets. Yet, these methods are much slower than the string trie and the FB-trie. For $k \leq 2$ and DNA data, they are also significantly slower than neighborhood generation methods.

It is noteworthy, that the BKT is a strictly sublinear method: given that $b = 20$, the BKT is $O\left(N^{0.69}\right)$ time for $k = 1$ and $O\left(N^{0.87}\right)$ time for $k = 2$ [Baeza-Yates and Navarro 1998]. In practice, however, the BKT is one of the slowest methods with very low filtering efficiency (see Table VIII, p. 62).

Both the multi-pivot modification of the BKT and the hybrid of the BKT and the AESA are slower than the BKT for all bucket sizes $b$ used in our experiments. Therefore, we do not present performance measures of these methods.

The retrieval time of all implemented methods grows exponentially with $k$ and most methods quickly degrade to sequential searching. According to data in Table VII, the growth rate of retrieval time for most methods that are not variants of neighborhood generation can be roughly approximated by $10^k$. For DNA-20, the retrieval time of the unigram frequency vector trie and the length-divided inverted $q$-gram grows approximately as $3^k$. Note that in the case of the string trie, the real growth rate is much lower than that implied by the theoretical upper bound of $O\left(n^{k+1}|\Sigma|^k\right)$.

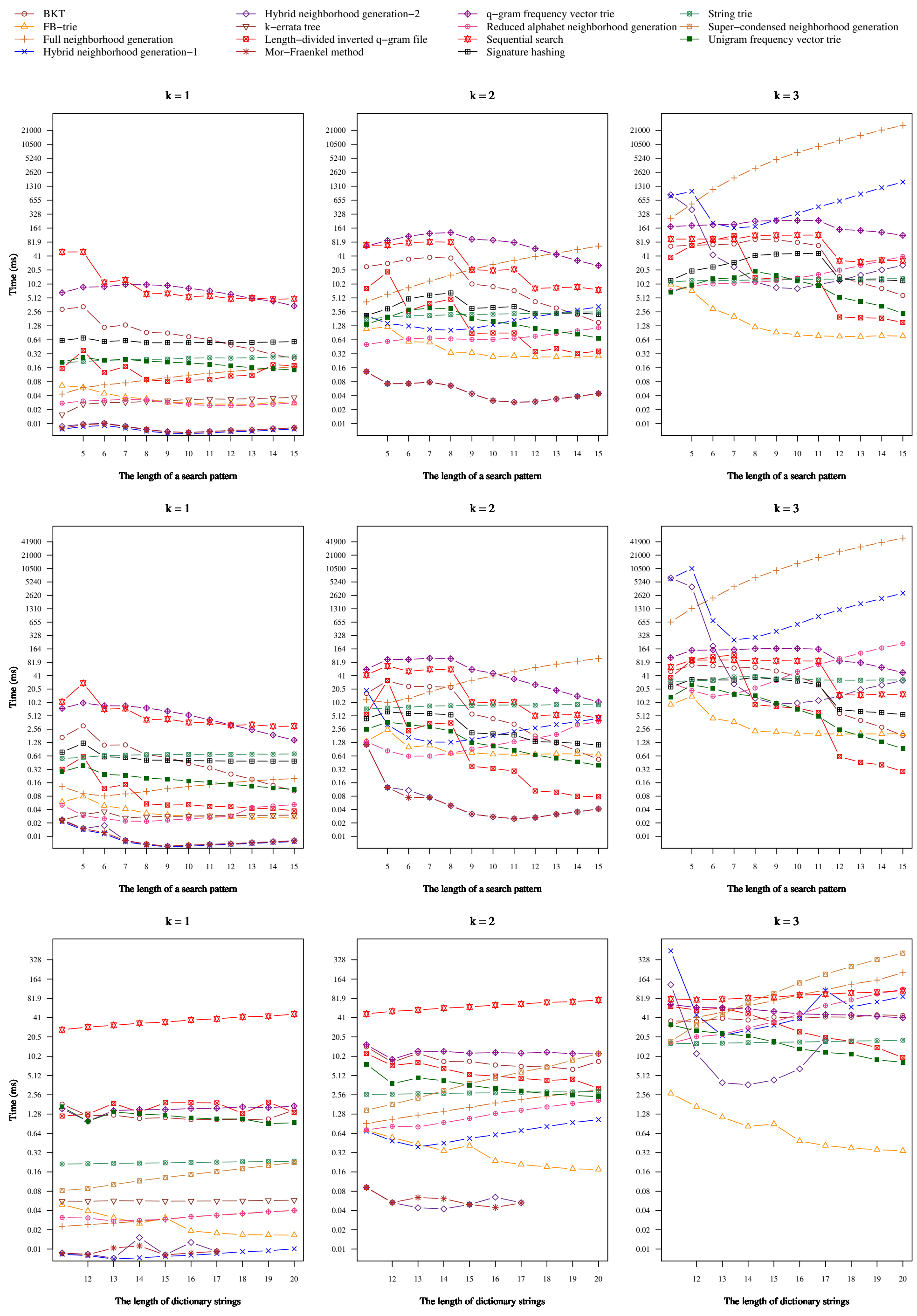
FB-trie has one of the slowest growth rates for all datasets, while full neighborhood generation has the fastest growth rate even for DNA data. Retrieval time of reduced alphabet generation exhibits a slower growth with $k$. In the case of natural languages and sparse dictionaries, reduced alphabet generation is faster than full neighborhood generation by a factor approximately equal to $(|\Sigma|/|\sigma|)^k$.

*8.3.3. Relationship between Retrieval Time and String Length.* To study the relationship between the retrieval time and the string length, we carry out experiments with DNA-Var dictionaries and the 0.2-million-string dictionary from ClueWeb09. Despite the differences in test scenarios and in alphabet sizes between the two data sets, there are similar patterns in both cases (see Figure 11, p. 60).

First of all, it can be seen that the retrieval time of full and super-condensed neighborhood generation is increasing as the string length increases, because the retrieval time of both methods is dominated by the time required to generate a neighborhood and search its elements. Note that, if $k$ is small in relation to the string length, the super-condensed neighborhood is not significantly smaller than the respective full neighborhood (see Table II, p. 28).

The filtering efficiency of Mor-Fraenkel method and hybrid neighborhood generation depends on the number of strings from intersections among the pattern $l$-deletion neighborhoods ($l \leq k$) and $l$-deletion neighborhoods of dictionary strings (see Appendix C.3.2 and Appendix C.3.3 for analytical justification). Because both a sub-

Fig. 11: The relationship between retrieval time and string length: the first row represents synthetic English data, the second row represents ClueWeb09 data, the third row represents DNA data



*Note:* Logarithmic scale on ax Y.

set of ClueWeb09 strings shorter than 5 characters and a set of short DNA sequences are dense, the number of such strings is larger. Therefore, these methods usually have longer retrieval time for small pattern sizes.

As the pattern length increases, the filtering efficiency increases as well. On the other hand, the size of the $l$-deletion neighborhood rapidly grows. As a result of these counter-balancing trends, retrieval time decreases to a certain point, where it achieves a minimum. Beyond this point, retrieval time is dominated by the size of the $l$-deletion neighborhood, which grows with the string length. Similar counter-balancing effects are observed in the case of reduced-alphabet neighborhood generation and natural language data. Note that this trend is more visible for $k > 1$.

The retrieval time of all sequence-based filtering methods generally decreases as the string length increases. We suppose that retrieval time is dominated by the cost of the checking step, which depends on the filtering efficiency. For a constant $k$, the filtering efficiency is better for longer strings, because a search algorithm uses longer subsequences to filter out non-matching strings.

The retrieval time of the string trie virtually does not depend on the lengths of patterns and dictionary strings. Note that if we search for patterns $p$ and $pv$ in the same dictionary, every node visited during the search for $p$ is also visited during the search for $pv$. In the case of ClueWeb09 and synthetic English data, this does not result in significantly worse retrieval time for long patterns. Yet, in the case of the FB-trie retrieval time is better if patterns and/or dictionary strings are longer.

To better understand this phenomenon, we have carried out experiments that include very short patterns (containing 2-3 characters). In addition to retrieval time, we have also measured the average number of nodes (including internal ones) visited at search time. As the pattern length increases from 2 to 15, the number of nodes visited in the string trie increases moderately, whereas the number of nodes visited in the FB-trie decreases sharply (by more than an order of magnitude).

Therefore, we conjecture that the retrieval time of the string trie is dominated by traversing nodes close to the root, where the average number of children is large. An increase in the pattern length does not lead to a significant increase in the number of visited nodes, because the number of child nodes quickly decreases as the node level increases. Searching in the FB-trie involves a step that filters out non-matching strings using a half of the pattern and the maximal allowed distance equal to $\lfloor k/2 \rfloor$. The efficiency of this step increases with the pattern length.

The case of DNA-Var data is similar to that of the natural language data. The first 10 levels of tries built for 10 different dictionaries are almost identical (the majority of nodes have exactly four children), and the retrieval time of the string trie is apparently dominated by traversal of nodes at the first 10 levels. The retrieval time of the FB-trie and the number of visited nodes are significantly lower for longer patterns and strings, because the filtering step works better with longer patterns.

*8.3.4. Filtering efficiency.* Experimental measurements of the filtering efficiency are presented in Table VIII. We divide all filtering methods into two groups: neighborhood generation filtering methods and the other filtering methods.

The first group includes: reduced alphabet generation, hybrid neighborhood generation methods, and Mor-Fraenkel method. The retrieval time of these methods is sublinear for one or more datasets. A disadvantage of all filtering neighborhood-generation methods is that the same string can be verified several times during the checking step. As a result, filtering efficiency can have negative values. Negative filtering efficiency is observed for hybrid neighborhood generation methods, $k = 3$, and all datasets except

Table VIII: Filtering efficiency

| | $k = 1$ | | | $k = 2$ | | | $k = 3$ | | |
|---|---|---|---|---|---|---|---|---|---|
| **Number of dictionary strings** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** |
| **English synthetic** | | | | | | | | | |
| BKT | 0.9278 | 0.9507 | 0.9708 | 0.6892 | 0.7183 | 0.7846 | 0.446 | 0.457 | 0.543 |
| Hybrid neighborhood generation-1 | 0.9999 | 0.9999 | | 0.9596 | 0.9896 | | -14.8 | -2.44 | |
| Hybrid neighborhood generation-2 | 0.9999 | | | 0.9996 | | | 0.730 | | |
| Length-divided inverted $q$-gram file | 0.9994 | 0.9996 | 0.9997 | 0.9866 | 0.9891 | 0.9918 | 0.890 | 0.901 | 0.919 |
| Mor-Fraenkel method | 0.9999 | | | 0.9996 | | | | | |
| $q$-gram frequency vector trie | 0.9613 | 0.9647 | 0.9675 | 0.5286 | 0.5153 | 0.5210 | 0.115 | 0.090 | 0.090 |
| Reduced alphabet neighborhood generation | 0.9983 | 0.9989 | 0.9993 | 0.9748 | 0.9821 | 0.9881 | 0.691 | 0.742 | 0.820 |
| Signature hashing | 0.9907 | 0.9902 | 0.9891 | 0.9163 | 0.9099 | 0.9002 | 0.676 | 0.659 | 0.635 |
| Unigram frequency vector trie | 0.9985 | 0.9987 | 0.9988 | 0.9865 | 0.9868 | 0.9877 | 0.934 | 0.931 | 0.938 |
| **Russian synthetic** | | | | | | | | | |
| BKT | 0.9360 | 0.9567 | 0.9754 | 0.7168 | 0.7411 | 0.8165 | 0.484 | 0.497 | 0.598 |
| Hybrid neighborhood generation-1 | 0.9999 | 0.9999 | | 0.9422 | 0.9855 | | -31.2 | -5.62 | |
| Hybrid neighborhood generation-2 | 0.9999 | | | 0.9996 | | | 0.555 | | |
| Length-divided inverted $q$-gram file | 0.9997 | 0.9997 | 0.9998 | 0.9941 | 0.9941 | 0.9952 | 0.943 | 0.943 | 0.951 |
| Mor-Fraenkel method | 0.9999 | | | 0.9996 | | | | | |
| $q$-gram frequency vector trie | 0.9771 | 0.9757 | 0.9791 | 0.6091 | 0.6037 | 0.6109 | 0.157 | 0.153 | 0.154 |
| Reduced alphabet neighborhood generation | 0.9993 | 0.9993 | 0.9995 | 0.9885 | 0.9893 | 0.9915 | 0.848 | 0.855 | 0.884 |
| Signature hashing | 0.9902 | 0.9902 | 0.9896 | 0.9076 | 0.9052 | 0.9010 | 0.659 | 0.654 | 0.637 |
| Unigram frequency vector trie | 0.9992 | 0.9993 | 0.9993 | 0.9917 | 0.9918 | 0.9923 | 0.958 | 0.955 | 0.960 |
| **ClueWeb09** | | | | | | | | | |
| BKT | 0.8876 | 0.9230 | 0.9555 | 0.6511 | 0.6834 | 0.7419 | 0.435 | 0.443 | 0.543 |
| Hybrid neighborhood generation-1 | 0.9999 | 0.9999 | | 0.9608 | 0.9894 | | -14.1 | -3.26 | |
| Hybrid neighborhood generation-2 | 0.9999 | | | 0.9988 | | | -1.75 | | |
| Length-divided inverted $q$-gram file | 0.9976 | 0.9988 | 0.9995 | 0.9595 | 0.9704 | 0.9803 | 0.768 | 0.791 | 0.851 |
| Mor-Fraenkel method | 0.9999 | | | 0.9988 | | | | | |
| $q$-gram frequency vector trie | 0.8902 | 0.9103 | 0.9334 | 0.3746 | 0.4214 | 0.4622 | 0.070 | 0.096 | 0.142 |
| Reduced alphabet neighborhood generation | 0.9977 | 0.9988 | 0.9994 | 0.9774 | 0.9866 | 0.9923 | 0.749 | 0.826 | 0.913 |
| Signature hashing | 0.9837 | 0.9877 | 0.9874 | 0.9170 | 0.9232 | 0.9257 | 0.701 | 0.712 | 0.726 |
| Unigram frequency vector trie | 0.9954 | 0.9969 | 0.9979 | 0.9685 | 0.9755 | 0.9809 | 0.872 | 0.887 | 0.922 |
| **DNA-11** | | | | | | | | | |
| BKT | 0.9483 | 0.9706 | 0.9804 | 0.6785 | 0.7559 | 0.7922 | 0.347 | 0.435 | 0.481 |
| Hybrid neighborhood generation-1 | 0.9998 | 0.9999 | | 0.9890 | 0.9948 | | 0.465 | 0.803 | |
| Hybrid neighborhood generation-2 | 0.9998 | | | 0.9941 | | | 0.633 | | |
| Length-divided inverted $q$-gram file | 0.9963 | 0.9965 | 0.9971 | 0.8990 | 0.9011 | 0.9080 | 0.549 | 0.546 | 0.569 |
| Mor-Fraenkel method | 0.9998 | | | 0.9941 | | | | | |
| $q$-gram frequency vector trie | 0.9739 | 0.9744 | 0.9722 | 0.7484 | 0.7336 | 0.7270 | 0.328 | 0.318 | 0.302 |
| Reduced alphabet neighborhood generation | 0.9980 | 0.9987 | 0.9994 | 0.9731 | 0.9779 | 0.9912 | 0.657 | 0.735 | 0.870 |
| Super-condensed neighborhood generation | 0.9999 | 0.9999 | 0.9999 | 0.9990 | 0.9991 | 0.9991 | 0.985 | 0.985 | 0.989 |
| Unigram frequency vector trie | 0.9556 | 0.9532 | 0.9447 | 0.8491 | 0.8397 | 0.8156 | 0.671 | 0.651 | 0.628 |
| **DNA-20** | | | | | | | | | |
| BKT | 0.9740 | 0.9853 | 0.9917 | 0.8180 | 0.8576 | 0.8922 | 0.533 | 0.596 | 0.651 |
| Hybrid neighborhood generation-1 | 0.9998 | 0.9999 | | 0.9790 | 0.9948 | | -0.96 | 0.508 | |
| Length-divided inverted $q$-gram file | 0.9978 | 0.9980 | 0.9978 | 0.9949 | 0.9950 | 0.9948 | 0.960 | 0.958 | 0.958 |
| $q$-gram frequency vector trie | 0.9963 | 0.9962 | 0.9963 | 0.9413 | 0.9377 | 0.9404 | 0.735 | 0.733 | 0.740 |
| Reduced alphabet neighborhood generation | 0.9999 | 0.9999 | 0.9999 | 0.9988 | 0.9996 | 0.9995 | 0.994 | 0.995 | 0.995 |
| Super-condensed neighborhood generation | 0.9999 | | | 0.9999 | | | 0.999 | | |
| Unigram frequency vector trie | 0.9859 | 0.9854 | 0.9859 | 0.9477 | 0.9446 | 0.9477 | 0.858 | 0.869 | 0.868 |

*Notes:* Filtering efficiency is calculated as $1-N_{check}/N$, where $N_{check}$ is a number of verifications performed during the checking step. M stands for million entries.

DNA-11. Negative efficiency indicates that filtering is not working properly and that the search algorithm is not better than sequential searching.

The second group comprises: the length-divided inverted $q$-gram file, signature hashing, the unigram frequency vector trie, the $q$-gram frequency vector trie, and the BKT. During the checking step, none of these methods verifies the same string more than once. Their retrieval times are roughly proportional to $N$.

According to experimental data, filtering efficiency correlates with retrieval time. The most efficient filtering methods are reduced alphabet neighborhood generation, the Mor-Fraenkel method (for $k \leq 2$), hybrid neighborhood generation methods (for

$k \leq 2$), the unigram frequency vector trie, and the length-divided $q$-gram file. They also have the highest filtering efficiency. The two slowest filtering methods – the BKT and the $q$-gram frequency vector trie – have low filtering efficiency.

It should be noted that filtering efficiency alone does not define retrieval performance. Efficiency of the method depends on many factors, which include verification cost and utilization of processor caches. For example, in the case of the synthetic Russian dataset and $k \leq 2$, length-divided inverted $q$-gram file has better filtering efficiency than the unigram frequency vector trie. Yet, for $N = 3.2 \times 10^6$, the length-divided inverted $q$-gram file is outperformed by the unigram frequency vector trie.

The checking step of both methods involves verifying whether the Levenshtein distance between a candidate string and the search pattern does not exceed $k$. Nonetheless, the unigram frequency vector trie combines strings into buckets, which allows for more efficient verification. An opposite case is observed for DNA-20 and $k = 3$. The unigram frequency vector has slightly shorter retrieval time than reduced alphabet generation, even though the latter has better filtering efficiency and uses a cheaper verification algorithm, which is essentially based on the Hamming distance.

*8.3.5. Index Size.* Index size statistics are presented in Table IX. All methods are divided into three groups: methods with sublinear, linear, and superlinear index overhead (the arithmetic difference between the total size of the index and the size of the original dictionary). Note that Table IX reports the full index size as opposed to the index overhead.

The first group includes the BKT, the unigram frequency vector trie, and the $q$-gram frequency vector trie. These methods have very small indices: for $N = 3.2 \times 10^6$, the size of the auxiliary data structure (excluding the strings themselves) is in the range from less than one percent to 43 percent of the dictionary size. By varying parameters of these methods, e.g., by decreasing the size of a frequency vector, it is possible to make the index overhead very small with respect to the dictionary size. Except for the unigram frequency vector trie, these methods are not very efficient.

In the case of natural language data and short DNA sequences, reduced alphabet generation has also a sublinear index overhead. This is a more efficient method than the BKT, the unigram and $q$-gram frequency vector trie, but its index overhead is also higher. One of the reasons is that reduced alphabet neighborhood generation employs a hash table with a load factor roughly equal to 0.5. A more compact representation is achievable if collisions are resolved with chaining. In the case of static dictionaries, further space savings are possible (see Section 4.1, p. 16).

The second group of methods includes full neighborhood generation, the string trie, the FB-trie, and the length-divided $q$-gram file. The overhead of full neighborhood generation is $\beta N$ bytes. In our implementation $\beta \approx 10$. Our implementation of super-condensed neighborhood generation is suboptimal in terms of space: it creates an inverted file of all substrings obtained from dictionary strings by deleting $i_1$ characters on the left and $i_2$ characters on the right (for all $i_1 + i_2 \leq 2k$). A more space-economical representation is possible with the help of (compressed) suffix trees or arrays.

The path-compressed string trie has 20-90 percent overhead. The FB-trie index consists of two string tries: one built over regular strings and the other built over reversed strings. It uses about twice the space of the regular trie. The FB-trie is the fastest method among those with linear and sublinear indices.

Static dictionaries can be represented as a succinct trie that uses space close to the information theoretic lower bound [Benoit et al. 2005]. In most cases, the resulting data structure uses less space than the original dictionary: i.e., its index overhead is negative. If it is not necessary to retrieve associated data, such as string identifiers, the string trie can be represented by a direct acyclic word graph, which in many cases also has a negative index overhead [Mihov and Schulz 2004].

The overhead of inverted $q$-gram file is $O(\alpha_{\text{comp}} \cdot \lambda NP)$, where the compression ratio $\alpha_{\text{comp}}$ depends on the type of $q$-gram and its size $q$. The best compression is achieved for $q = 2$. By increasing $q$ from 2 to a dataset-dependent optimal value (see Table VI, p. 48), we only slightly increase the size of the index, but significantly decrease retrieval time. By further increasing $q$, the index size continues to grow, while retrieval time improves only marginally.

The compression ratio $\alpha_{\text{comp}}$ varies across datasets: it is less than 0.5 for DNA data and is approximately equal to 0.6 for synthetic natural language data. For ClueWeb09, the best compression ratio is 0.84. Note that the data in Table IX includes the space used by the set of short strings that are searched for sequentially. This additional overhead is small (less than 2 percent of the dictionary size) for all datasets except ClueWeb09, for which it is equal to 23 percent of the dictionary size.

Table IX: Index Size

| | **English synthetic** | | | **Russian synthetic** | | | **ClueWeb09** | | | **DNA-11** | | | **DNA-20** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Number of dictionary strings** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** | **0.2M** | **0.8M** | **3.2M** |
| BKT | 2MB | 8MB | 35MB | 2MB | 9MB | 38MB | 2MB | 7MB | 29MB | 2MB | 10MB | 39MB | 4MB | 17MB | 67MB |
| | 104% | 104% | 104% | 103% | 103% | 103% | 104% | 104% | 104% | 107% | 107% | 107% | 104% | 104% | 104% |
| FB-trie | 6MB | 25MB | 95MB | 7MB | 26MB | 101MB | 6MB | 22MB | 90MB | 7MB | 26MB | 84MB | 10MB | 40MB | 155MB |
| | 330% | 307% | 282% | 316% | 298% | 278% | 373% | 347% | 324% | 307% | 279% | 230% | 262% | 252% | 242% |
| Full neighborhood generation | 5MB | 20MB | 82MB | 5MB | 21MB | 84MB | 4MB | 18MB | 76MB | 5MB | 21MB | 85MB | 7MB | 28MB | 112MB |
| | 253% | 248% | 243% | 237% | 235% | 232% | 300% | 286% | 272% | 231% | 231% | 231% | 175% | 175% | 175% |
| Hybrid neighborhood generation-1 | 48MB | 193MB | | 54MB | 217MB | | 26MB | 159MB | | 26MB | 63MB | | 117MB | 468MB | |
| | 2448% | 2379% | | 2483% | 2447% | | 1746% | 2472% | | 1156% | 687% | | 2928% | 2923% | |
| Hybrid neighborhood generation-2 | 281MB | | | 440MB | | | 142MB | | | 99MB | | | | | |
| | 14288% | | | 20070% | | | 9485% | | | 4318% | | | | | |
| $k$-errata tree | 151MB | 629MB | | 169MB | 692MB | | 107MB | 469MB | | 164MB | 550MB | | 289MB | | |
| | 7696% | 7752% | | 7682% | 7809% | | 7161% | 7277% | | 7176% | 6010% | | 7211% | | |
| Length-divided inverted $q$-gram file | 11MB | 32MB | 120MB | 14MB | 39MB | 136MB | 9MB | 33MB | 129MB | 6MB | 25MB | 98MB | 12MB | 46MB | 183MB |
| | 563% | 400% | 356% | 627% | 445% | 375% | 616% | 519% | 462% | 269% | 268% | 268% | 288% | 286% | 286% |
| Mor-Fraenkel method | 281MB | | | 440MB | | | 142MB | | | 99MB | | | | | |
| | 14288% | | | 20070% | | | 9485% | | | 4318% | | | | | |
| $q$-gram frequency vector trie | 4MB | 15MB | 48MB | 5MB | 17MB | 56MB | 3MB | 11MB | 40MB | 3MB | 9MB | 37MB | 6MB | 19MB | 69MB |
| | 228% | 180% | 143% | 236% | 192% | 154% | 204% | 167% | 145% | 112% | 103% | 101% | 140% | 119% | 107% |
| Reduced alphabet neighborhood generation | 5MB | 14MB | 46MB | 5MB | 16MB | 58MB | 4MB | 17MB | 70MB | 4MB | 11MB | 39MB | 12MB | 46MB | 178MB |
| | 238% | 174% | 136% | 242% | 181% | 160% | 280% | 264% | 251% | 162% | 124% | 107% | 290% | 286% | 278% |
| Signature hashing | 3MB | 10MB | 37MB | 4MB | 11MB | 40MB | 3MB | 8MB | 31MB | | | | | | |
| | 165% | 121% | 111% | 160% | 120% | 111% | 182% | 124% | 112% | | | | | | |
| String trie | 3MB | 13MB | 49MB | 4MB | 14MB | 52MB | 3MB | 11MB | 45MB | 4MB | 13MB | 42MB | 5MB | 20MB | 77MB |
| | 169% | 157% | 145% | 165% | 154% | 143% | 188% | 175% | 163% | 153% | 139% | 115% | 131% | 126% | 121% |
| Super-condensed neighborhood generation | | | | | | | | | | 184MB | 573MB | | 641MB | | |
| | | | | | | | | | | 8055% | 6257% | | 16014% | | |
| Unigram frequency vector trie | 3MB | 11MB | 38MB | 4MB | 13MB | 45MB | 2MB | 9MB | 35MB | 2MB | 9MB | 37MB | 4MB | 16MB | 64MB |
| | 168% | 133% | 114% | 189% | 149% | 124% | 160% | 136% | 124% | 100% | 100% | 100% | 101% | 100% | 100% |

*Notes:* Percentage represents the ratio of the index size to the lexicon size.

Table X: Ratio of retrieval time for transposition-aware modification to retrieval time for transposition-unaware modification.

| | English synthetic | | | Russian synthetic | | | ClueWeb09 | | |
|---|---|---|---|---|---|---|---|---|---|
| | $k = 1$ | $k = 2$ | $k = 3$ | $k = 1$ | $k = 2$ | $k = 3$ | $k = 1$ | $k = 2$ | $k = 3$ |
| BKT | 1.28 | 1.73 | 2.90 | 1.17 | 1.48 | 2.58 | 1.40 | 1.92 | 2.10 |
| FB-trie | 1.09 | 1.19 | 1.35 | 1.08 | 1.20 | 1.10 | 1.10 | 1.16 | 1.33 |
| Full neighborhood generation | 1.01 | 1.03 | 1.04 | 1.01 | 1.02 | 0.95 | 1 | 1.02 | 1.03 |
| Hybrid neighborhood generation-1 | 1 | 1.01 | 1.04 | 1 | 1.02 | 1 | 1.02 | 1.29 | 1.04 |
| Hybrid neighborhood generation-2 | 1.01 | 1.02 | 1.02 | 1 | 1 | 1.01 | 1.01 | 1.05 | 1.03 |
| $k$-errata tree | 1 | | | 0.98 | | | 0.99 | | |
| Length-divided inverted $q$-gram file | 1.14 | 1.21 | 1.33 | 1.14 | 1.22 | 1.36 | 1.21 | 1.21 | 1.28 |
| Mor-Fraenkel method | 1 | 1 | | 1 | 1 | | 1.01 | 1.05 | |
| $q$-gram frequency vector trie | 4.42 | 7.06 | 3.61 | 4 | 5.52 | 4.79 | 4.56 | 4.99 | 2.96 |
| Reduced alphabet neighborhood generation | 1.13 | 1.23 | 1.56 | 1.12 | 1.23 | 1.34 | 1.08 | 1.13 | 1.28 |
| **Sequential search** | **1.25** | **2.27** | **3.37** | **1.22** | **1.95** | **3.23** | **1.77** | **2.85** | **2.58** |
| Signature hashing | 1.18 | 1.92 | 2.95 | 1.02 | 1.50 | 2.54 | 2.64 | 2.38 | 2.50 |
| String trie | 1.19 | 1.15 | 1.14 | 1.74 | 1.21 | 1.10 | 1.15 | 1.15 | 1.13 |
| Unigram frequency vector trie | 1.52 | 1.58 | 3.23 | 0.87 | 1.29 | 2.68 | 1.39 | 2 | 1.89 |

The third group represents superlinear methods and includes the Mor-Fraenkel method, hybrid neighborhood generation methods, and the $k$-errata tree. The Mor-Fraenkel method relies on indexing of $l$-deletion dictionaries ($l \leq k$). Whenever the index fits into the main memory, the Mor-Fraenkel method outperforms all other methods. However, this is possible only for small dictionaries and $k \leq 2$. In our implementation, the index is more than 40 times larger than the size of the dictionary in the case of DNA data and is more than 140 times larger in the case of natural language data. A transducer-based variant of Mor-Fraenkel method has a smaller index for $k = 1$, but it has not been verified whether the transducer-based index is smaller for $k > 1$.

One approach to reduce index size consists in combining Mor-Fraenkel method with full neighborhood generation. However, this method has not worked well for $k = 3$.

Some of the best retrieval times are also achieved by the $k$-errata tree. Yet, its index is so large that the method cannot be tested for $k > 1$.

*8.3.6. Cost of Including Transpositions as Basic Edit Operations.* From Table X it follows that counting the transposition as a single edit operation does have a significant cost for many methods.

The most "transposition-sensitive" method is the $q$-gram frequency vector trie: the difference in retrieval time between a transposition-aware and a transposition-unaware modification is 3-7 fold. The search algorithm of this method is based on Inequality 11, p. 13, which is not very efficient in the case of transposition-aware searching.

The least transposition-sensitive methods are the Mor-Fraenkel method, hybrid neighborhood generation methods, the $k$-errata tree, and full neighborhood generation. The transposition-aware versions of reduced alphabet neighborhood generation, the length-divided inverted $q$-gram file, the string trie, and the FB-trie are 1.1-1.7 times slower than respective transposition-unaware modifications.

## 9. CONCLUSIONS

In this paper we survey state of the art indexing methods for approximate dictionary searching, which we classify into direct methods and sequence-based filtering methods. The taxonomy is explained in Section 3 and outlined in Figure 2, p. 14. We present experimental results for maximum allowed edit distance $k = 1, 2, 3$ and several data sets that include natural language dictionaries and dictionaries of DNA sequences (see

Section 8.1.2, p. 46). The summary of implemented methods is given in Table V, p. 48; parameters of implemented methods are listed in Table VI, p. 48.

Because of theoretical advances and technological breakthroughs in computer hardware, approximate dictionary queries can be answered very quickly. For all datasets except DNA-11 and ClueWeb09, a query can be answered within 2 milliseconds (on average). Implemented methods are up to four orders of magnitude faster than sequential searching. However, retrieval time grows exponentially with the maximum allowed edit distance: as $k$ increases, all the methods degrade to sequential searching.

Another important lesson that we have learned from our experiments is that some datasets are less amenable to efficient indexing than others. In particular, ClueWeb09 contains many short words and is prefix-dense (ClueWeb09 includes almost all possible three-letter words). As a result, the retrieval time of both string tries and FB-tries for ClueWeb09 is an order of magnitude longer than for natural language dictionaries and DNA-20. Another difficult dataset is DNA-11, where some dictionaries are dense, i.e. contain almost all existing strings of a given length. Therefore, in the case of DNA-11, it is not possible to significantly outperform full neighborhood generation. Obviously, DNA-11 is also a prefix-dense dataset, which inhibits performance of prefix trees such as string tries and FB-tries.

In our opinion, there are two most important types of search methods: neighborhood generation and string tries. These are conceptually similar methods (see Section 6.2.2, p. 27), which also have comparable retrieval time. For DNA-11, full neighborhood generation has some of the best retrieval times and outperforms string tries for $k \leq 2$. In the case of natural language data, full neighborhood generation is not especially efficient. Yet, its modification – reduced alphabet generation – has a performance equivalent to that of tries.

There are two methods that are superlinear in terms of index size: the Mor-Fraenkel method and the $k$-errata tree. the Mor-Fraenkel method outperforms the string trie, full neighborhood generation, and reduced alphabet generation for all dictionaries, where it is possible to fit the index of Mor-Fraenkel method into the memory. The retrieval time of the $k$-errata tree is comparable with the retrieval time of the FB-trie. However, the index of the $k$-errata tree is the largest among implemented methods: we believe that the current version of the method is impractical for $k > 1$. The Mor-Fraenkel method is more space efficient, but its application is also limited to small dictionaries and small $k$. On the other hand, a hybrid method, FB-trie, which blends tries with pattern partitioning, achieves excellent retrieval performance using linear space. Evidently, better compromise solutions are yet to be discovered. We conjecture that these solutions would combine neighborhood generation, tries, and pattern partitioning.

Sequence-based filtering methods are not very fast. The unigram frequency vector trie is a reasonably efficient method with a very small index overhead, which is up to two orders of magnitude faster than sequential searching. Yet, it is significantly slower than the FB-trie or the Mor-Fraenkel method. A very similar method, the $q$-gram frequency vector trie, is almost always slower than the unigram frequency vector. It is especially inefficient for natural language data. For $k > 1$, the length-divided inverted $q$-gram file, which employs partitioning into exact searching, is also slower than the unigram frequency trie in most cases. In addition, it has much larger index overhead.

In comparison to other methods, metric space methods also appear to be slow, which confirms earlier conclusions [Baeza-Yates and Navarro 1998]. Our implementation of the BKT is about 3-15 times faster than sequential searching for $k = 2$ and is only about 2-3 times faster than sequential searching for $k = 3$. We have also implemented the multi-pivot modification of the BKT as well as a hybrid of the BKT and the AESA. Neither method was more efficient than the BKT. Fredriksson [2007] reports that almost a ten-fold improvement can be achieved in BKT retrieval times by applying a bit-parallel algorithm that computes several distances simultaneously. Yet, we believe that this improvement is mainly due to using SIMD extensions (Single Instruction Multiple Data), which are available on Intel and AMD64 processors (see a discussion in Section 6.3.3, p. 35).

Approximate string searching is an area of active research, where a lot of practical and theoretical problems remain unsolved:

— Belazzougui [2009] demonstrates that there exists a hybrid method with the index size proportional to the dictionary size that answers approximate query in time proportional to the pattern size for $k = 1$. How does this result scale to a larger $k$?

— From [Ukkonen 1993] it follows that the average number of trie nodes visited during searching can be upper bounded by $O(|\Sigma|^k n^{k+1})$. This estimate predicts a much faster growth (depending on $k$) than that observed in our experiments, where retrieval time increases approximately as $10^k$. As demonstrated by Navarro and Baeza-Yates [2000], this estimate can be improved, but the improvement is not significant (see Appendix C.2.1, p. 79). It would be useful to derive a tighter upper bound for the average number of visited nodes.

— The FB-trie uses a pair of string tries to satisfy a query. The search algorithm is a two step procedure that divides the pattern into two parts and searches for the first part with $t \leq k/2$ errors and for the second part with $k - t$ errors. In most practical situations, the number of child nodes in a trie decreases quickly as the node level increases. Therefore, the computational cost is dominated by the first step of the search algorithm, where the maximum allowed edit distance is only $t \leq k/2$. However, we are not aware of any rigorous analysis that supports this conjecture.

— In this survey we show that there is a variety of efficient approximate search methods, where similarity is expressed in terms of Levenshtein or Damerau-Levenshtein distance. Brill and Moore [2000] show that, for the purpose of spell-checking, higher correction accuracy is achieved with the extended edit distance that allows for weighted substitutions of arbitrary substrings (see Section 2.2).[31] Efficiency of the methods that use the extended edit distance is less studied. Because the extended edit distance is computationally more expensive, optimization of these methods is an important research venue.

## ACKNOWLEDGMENTS

Before all, I express my gratitude to Anna Belova who was the first technical editor of this paper. It is her help and encouragement that made this work possible.

I especially thank Gonzalo Navarro for sharing an implementation of the lazy DFA and making a lot of valuable suggestions and corrections. I am thankful to anonymous reviewers who suffered through earlier versions of the paper and provided important feedback.

I also thank Ricardo Baeza-Yates, Esko Ukkonen, Justin Zobel, Hans-Jörg Schek, and Aviezri Fraenkel for answering my questions about their work; Maxim Gubin, Oleg Khovayko, and Tetiana Zabolotnia for their comments on the paper. I am also grateful to the readers of my blog who verified the proof of Theorem C.1 as well as to an anonymous friend who retrieved several rare papers for me. Last but not least, I thank Igor Gorbatov for attracting my attention to the problem in the first place.

[31] Brill and Moore [2000] define similarity of strings $p$ and $s$ as the maximum probability of transforming one string into another. To compute this probability, they consider all partitions of $p$ and $s$ into substrings $\{p_i\}$ and $\{s_i\}$, respectively. Each partition defines an edit script. Probability associated with this script is calculated as a product of probabilities of replacing $p_i$ with $s_i$. Then probability is maximized over all possible partitions. This definition is equivalent to finding an optimal alignment, where the cost of replacing $p_i$ with $s_i$ is equal to the logarithm of corresponding probability.

## REFERENCES

AMIR, A., KESELMAN, D., LANDAU, G. M., LEWENSTEIN, M., LEWENSTEIN, N., AND RODEH, M. 2000. Text indexing and dictionary matching with one error. *Journal of Algorithms 37,* 2, 309–325.

ANGELL, R. C., FREUND, G. E., AND WILLETT, P. 1983. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management 19,* 4, 443–453.

BAEZA-YATES, R., CUNTO, W., MANBER, U., AND WU, S. 1994. Proximity matching using fixed-queries trees. In *Combinatorial Pattern Matching*, M. Crochemore and D. Gusfield, Eds. Lecture Notes in Computer Science Series, vol. 807. Springer Berlin / Heidelberg, 198–212.

BAEZA-YATES, R. AND GONNET, G. 1992. A new approach to text searching. *Communications of the ACM 35,* 10, 74–82.

BAEZA-YATES, R. AND NAVARRO, G. 1998. Fast approximate string matching in a dictionary. In *Proceedings of the SPIRE'98*. 14–22.

BAEZA-YATES, R. AND NAVARRO, G. 1999. Faster approximate string matching. *Algorithmica 23,* 2, 127–158.

BAEZA-YATES, R. AND SALINGER, A. 2005. Experimental analysis of a fast intersection algorithm for sorted sequences. In *String Processing and Information Retrieval*, M. Consens and G. Navarro, Eds. Lecture Notes in Computer Science Series, vol. 3772. Springer Berlin / Heidelberg, 13–24.

BAEZA-YATES, R. A. AND GONNET, G. H. 1990. All-against-all sequence matching: Preliminary version. http://www.dcc.uchile.cl/˜rbaeza/cv/reports.html.

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 '99. IEEE Computer Society, Washington, DC, USA, 16–23.

BARBAY, J., LÓPEZ-ORTIZ, A., AND LU, T. 2006. Faster adaptive set intersections for text searching. In *Experimental Algorithms*, C. Àlvarez and M. Serna, Eds. Lecture Notes in Computer Science Series, vol. 4007. Springer Berlin / Heidelberg, 146–157.

BECKMANN, N., KRIEGEL, H., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of 1990 ACM SIGMOD international conference on Management of data*. ACM, 322–331.

BELAZZOUGUI, D. 2009. Faster and space-optimal edit distance 1 dictionary. In *Combinatorial Pattern Matching*, G. Kucherov and E. Ukkonen, Eds. Lecture Notes in Computer Science Series, vol. 5577. Springer Berlin / Heidelberg, 154–167.

BENOIT, D., DEMAINE, E. D., MUNRO, J. I., RAMAN, R., RAMAN, V., AND RAO, S. S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4, 275–292.

BENTLEY, J. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18,* 9, 509–517.

BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H. P. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases*. Morgan Kaufmann, 28–39.

BERMAN, A. 1994. A new data structure for fast approximate matching. Technical report 1994-03-02, Department of Computer Science and Engineering, University of Washington.

BIEGANSKI, P., RIEDL, J., CARTIS, J., AND RETZEL, E. 1994. Generalized suffix trees for biological sequence data: applications and implementation. In *System Sciences, 1994. Vol.V: Biotechnology Computing, Proceedings of the Twenty-Seventh Hawaii International Conference*. Vol. 5. 35 –44.

BÖHM, C., BERCHTOLD, S., AND KEIM, D. A. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys 33,* 3, 322–373.

BOITSOV, L. 2004. Classification and experimental comparison of modern dictionary fuzzy search algorithms (in Russian). In *RCDL: Proceedings of the 6th Russian Conference on Digital Libraries*.

BOTELHO, F., PAGH, R., AND ZIVIANI, N. 2007. Simple and space-efficient minimal perfect hash functions. In *Algorithms and Data Structures*, F. Dehne, J.-R. Sack, and N. Zeh, Eds. Lecture Notes in Computer Science Series, vol. 4619. Springer Berlin / Heidelberg, 139–150.

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*. 286–293.

BURKHARD, W. AND KELLER, R. 1973. Some approaches to best-match file searching. *Communications of the ACM 16,* 4, 230–236.

BURKHARDT, S. AND KÄRKKÄINEN, J. 2002. One-gapped $q$-gram filter for Levenshtein distance. *In Lecture Notes in Computer Science 2373*, 225–234.

CÁNOVAS, R. AND NAVARRO, G. 2010. Practical compressed suffix trees. In *Experimental Algorithms*, P. Festa, Ed. Lecture Notes in Computer Science Series, vol. 6049. Springer Berlin / Heidelberg, 94–105.

CARTERETTE, B. AND CAN, F. 2005. Comparing inverted files and signature files for searching in a large lexicon. *Information Processing and Management 41,* 3, 613–633.

CHAN, H.-L., LAM, T.-W., SUNG, W.-K., TAM, S.-L., AND WONG, S.-S. 2006. Compressed indexes for approximate string matching. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*. Springer-Verlag, London, UK, 208–219.

CHÁVEZ, E., MARROQUÍN, J. L., AND NAVARRO, G. 2001. Fixed Queries Array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications 14,* 2, 113–135.

CHÁVEZ, E. AND NAVARRO, G. 2003. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters 85,* 1, 39–46.

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

CHO, J. AND RAJAGOPALAN, S. 2002. A fast regular expression indexing engine. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE Computer Society, 419–430.

CLAUDE, F., FARIÑA, A., AND NAVARRO, G. 2009. Re-Pair compression of inverted lists. *CoRR abs/0911.3318*.

COELHO, L. AND OLIVEIRA, A. 2006. Dotted Suffix Trees a structure for approximate text indexing. In *String Processing and Information Retrieval*, F. Crestani, P. Ferragina, and M. Sanderson, Eds. Lecture Notes in Computer Science Series, vol. 4209. Springer Berlin / Heidelberg, 329–336.

COLE, R., GOTTLIEB, L.-A., AND LEWENSTEIN, M. 2004. Dictionary matching and indexing with errors and don't cares. In *STOC '04: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. ACM, 91–100.

COLLINS, P. 2009. Definately* the most misspelled word in the english language (*it should be definitely). The Daily Record. June 15, 2009.
http://www.dailyrecord.co.uk/news/editors-choice/2009/06/15/definately-the-most-misspelled-word-in-the-english-language-it-should-be-definitely-86908-21441847/.

COVINGTON, M. A. 1996. An algorithm to align words for historical comparison. *Computational Linguistics 22,* 4, 481–496.

DACIUK, J., MIHOV, S., WATSON, B. W., AND WATSON, R. E. 2000. Incremental construction of minimal acyclic finite-state automata. *Comput. Linguist. 26,* 1, 3–16.

DAMERAU, F. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM 7,* 3, 171–176.

D'AMORE, R. J. AND MAH, C. P. 1985. One-time complete indexing of text: theory and practice. In *SIGIR '85: Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, USA, 155–164.

DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2001. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX '01: Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*. Springer-Verlag, London, UK, 91–104.

DIETZFELBINGER, M., KARLIN, A., MEHLHORN, K., MEYER AUF DER HEIDE, F., ROHNERT, H., AND TARJAN, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput. 23,* 4, 738–761.

DÖMÖLKI, B. 1968. A universal compiler system based on production rules. *BIT Numerical Mathematics 8*, 262–275.

DOSTER, W. 1977. Contextual postprocessing system for cooperation with a multiple-choice character-recognition system. *IEEE Trans. Comput. 26*, 1090–1101.

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.

ELIAS, P. 1974. Efficient storage and retrieval by content and address of static files. *J. ACM 21,* 2, 246–260.

FALOUTSOS, C. 1996. *Searching Multimedia Databases by Content*. Kluwer Academic Publisher.

FERRAGINA, P., MUTHUKRISHNAN, S., AND DE BERG, M. 1999. Multi-method dispatching: a geometric approach with applications to string matching problems. In *STOC '99: Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*. ACM, New York, NY, USA, 483–491.

FERRAGINA, P. AND VENTURINI, R. 2007. Compressed permuterm index. In *SIGIR '07: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, USA, 535–542.

FIGUEROA, K. AND FREDRIKSSON, K. 2007. Simple space-time trade-offs for AESA. In *Experimental Algorithms*. Springer Berlin, 229–241.

FISCHER, J., MÄKINEN, V., AND NAVARRO, G. 2008. An(other) entropy-bounded compressed suffix tree. In *CPM '08: Proceedings of the 19th annual symposium on Combinatorial Pattern Matching*. Springer-Verlag, Berlin, Heidelberg, 152–165.

FORD, G., HAUSE, S., LE, D. X., AND THOMA, G. R. 2001. Pattern matching techniques for correcting low confidence OCR words in a known context. In *Proceedings of the 8th SPIE International Conference on Document Recognition and Retrieval*. 241–249.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with $o(1)$ worst case access time. *J. ACM 31,* 3, 538–544.

FREDRIKSSON, K. 2007. Engineering efficient metric indexes. *Pattern Recogn. Lett. 28,* 1, 75–84.

FRIEDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM transactions on Mathematical Software 3,* 3, 209–226.

GAEDE, V. AND GÜNTHER, O. 1998. Multidimensional access methods. *ACM Computing Surveys 30,* 2, 170–231.

GIEGERICH, R., KURTZ, S., AND STOYE, J. 2003. Efficient implementation of lazy suffix trees. *Softw., Pract. Exper. 33,* 11, 1035–1049.

GILADI, E., WALKER, M. G., WANG, J. Z., AND VOLKMUTH, W. 2002. SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics 18,* 6, 873–877.

GLASS, J. R. 2003. A probabilistic framework for segment-based speech recognition. *Computer Speech & Language 17,* 2-3, 137 – 152. New Computational Paradigms for Acoustic Modeling in Speech Recognition.

GOLLAPUDI, S. AND PANIGRAHY, R. 2006. A dictionary for approximate string search and longest prefix search. In *CIKM '06: Proceedings of the 15th ACM International Conference on Information and Knowledge Management*. ACM, 768–775.

GORIN, R. E. 1971. SPELL: Spelling check and correction program. Online documentation. http://pdp-10.trailing-edge.com/decuslib10-03/01/43,50270/spell.doc.html.

GRAVANO, L., IPEIROTIS, P., JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., PIETARINEN, L., AND SRIVASTAVA, D. 2001. Using $q$-grams in a DBMS for approximate string processing. *IEEE Data Engineering Bulletin 24,* 4, 28–34.

GROSSI, R. AND LUCCIO, F. 1989. Simple and efficient string matching with $k$ mismatches. *Information Processing Letters 33,* 3, 113–120.

GROSSI, R. AND VITTER, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

GUSFIELD, D. 1999. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press.

GUTTMAN, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD international conference on Management of data*. ACM Press, 47–57.

GWEHENBERGER, G. 1968. Anwendung einer binären verweiskettenmethode beim aufbau von listen. *Elektronische Rechenanlagen 10*, 223–226.

HALL, P. AND DOWLING, G. 1980. Approximate string matching. *ACM Computing Surveys 12,* 4, 381–402.

HELLERSTEIN, J. AND PFEFFER, A. 1994. The RD-tree: An index structure for sets. Technical report No 1252, University of Wisconsin at Madison.

HOEFFDING, W. 1963. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association 58,* 301, 13–30.

HYYRÖ, H. 2003a. Bit-parallel approximate string matching algorithms with transposition. In *Proceedings of the 10th International Symposium on String Processing*. Springer-Verlag, 95–107.

HYYRÖ, H. 2003b. Practical methods for approximate string matching. Ph.D. thesis, Department of Computer Science, University of Tampere, Finland.

HYYRÖ, H. 2005. Bit-parallel approximate string matching algorithms with transposition. *Journal of Discrete Algorithms 3,* 2-4, 215–229.

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

JEONG, I.-S., PARK, K.-W., KANG, S.-H., AND LIM, H.-S. 2010. An efficient similarity search based on indexing in large dna databases. *Computational Biology and Chemistry 34,* 2, 131 – 136.

JOKINEN, P., TARHIO, J., AND UKKONEN, E. 1996. A comparison of approximate string matching algorithms. *Software Practice and Experience 26,* 12, 1439–1458.

JOKINEN, P. AND UKKONEN, E. 1991. Two algorithms for approximate string matching in static texts. *Lecture Notes in Computer Science 520*, 240–248.

KAHVECI, T., LJOSA, V., AND SINGH, A. K. 2004. Speeding up whole-genome alignment by indexing frequency vectors. *Bioinformatics 20,* 13, 2122–2134.

KAHVECI, T. AND SINGH, A. 2001. An efficient index structure for string databases. In *Proceedings of the 27th International Conference on Very Large Databases*. Morgan Kaufmann, 351–360.

KLOVSTAD, J. AND MONDSHEIN, L. 1975. The CASPERS linguistic analysis system. *IEEE Transactions on Acoustics, Speech and Signal Processing 23*, 118 – 123.

KNUTH, D. 1973. *The Art of Computer Programming. Sorting and Searching* 1st Ed. Vol. 3. Addison-Wesley.

KNUTH, D. 1997. *The Art of Computer Programming. Sorting and Searching* 2d Ed. Vol. 3. Addison-Wesley.

KONDRAK, G. 2003. Phonetic alignment and similarity. *Computers and the Humanities 37,* 3, 273–291.

KUENNING, G., GORIN, R. E., WILLISSON, P., BUEHRING, W., AND STEVENS, K. 1988. International spell: a fast screen-oriented spelling checker. Ispell.info – documentation of the Ispell. See also http://www.lasr.cs.ucla.edu/geoff/ispell.html.

KUKICH, K. 1992. Technique for automatically correcting words in text. *ACM Computing Surveys 24,* 2, 377–439.

KURTZ, S. 1996. Approximate string searching under weighted edit distance. In *Proceedings of the 3rd South American Workshop on String Processing*. Carleton University Press, 156–170.

KURTZ, S. 1999. Reducing the space requirement of suffix trees. *Softw. Pract. Exper. 29,* 13, 1149–1171.

LEVENSHTEIN, V. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR, 163,* 4, 845–848.

LOWRANCE, R. AND WAGNER, R. 1975. An extension of the string-to-string correction problem. *Journal of the ACM 22,* 2, 177–183.

MALY, K. 1976. Compressed tries. *Communications of the ACM 19,* 7, 409–415.

MANBER, U. AND MYERS, G. 1990. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 319–327.

MANBER, U. AND WU, S. 1994a. An algorithm for approximate membership checking with application to password security. *Information Processing Letters 50,* 4, 191 – 197.

MANBER, U. AND WU, S. 1994b. GLIMPSE: a tool to search through entire file systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. USENIX Association, 4–4.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23,* 2, 262–272.

MELICHAR, B. 1996. String matching with $k$ differences by finite automata. In *Proceedings of the International Congress on Pattern Recognition*. 256–260.

MICÓ, M. L., ONCINA, J., AND VIDAL-RUIZ, E. 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett. 15,* 1, 9–17.

MIHOV, S. AND SCHULZ, K. U. 2004. Fast approximate string search in large dictionaries. *Computational Linguistics, 30,* 4, 451–477.

MOCHIZUKI, H. AND HAYASHI, Y. 2000. An efficient retrieval algorithm of compound words using extendible hashing. *International Journal of Computer Processing of Oriental Languages 13,* 1, 15–33.

MOORE, T. AND EDELMAN, B. 2010. Measuring the perpetrators and funders of typosquatting. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*. Springer. To be published.

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.

MORRISON, D. R. 1968. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15,* 4, 514–534.

MUTH, R. AND MANBER, U. 1996. Approximate multiple string search. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*. Springer, 75–86.

MYERS, E. 1994. A sublinear algorithm for approximate keyword searching. *Algorithmica 12,* 4/5, 345–374.

MYERS, E. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM 46,* 3, 395–415.

NAVARRO, G. 1997a. Multiple approximate string matching by counting. In *Proceedings of the 4th South American Workshop on String Processing*, R. Baeza-Yates, Ed. Carleton University Press, Valparaiso, Chile, 95–111.

NAVARRO, G. 1997b. A partial deterministic automaton for approximate string matching. In *Proceedings of the 4th South American Workshop on String Processing*. Carleton University Press, 112–124.

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

NAVARRO, G. 2001b. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience 31,* 13, 1265–1312.

NAVARRO, G. AND BAEZA-YATES, R. 1998. A practical $q$-gram index for text retrieval allowing errors. *CLEI Electronic Journal 1,* 2, http://www.clei.cl.

NAVARRO, G. AND BAEZA-YATES, R. 2000. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms, 1,* 1, 205–239.

NAVARRO, G., BAEZA-YATES, R., SUTINEN, E., AND TARHIO, J. 2001. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin 24,* 4, 19–27.

NAVARRO, G., PAREDES, R., AND CHÁVEZ, E. 2002. t-Spanners as a data structure for metric space searching. In *String Processing and Information Retrieval*, A. Laender and A. Oliveira, Eds. Lecture Notes in Computer Science Series, vol. 2476. Springer Berlin / Heidelberg, 195–200.

NAVARRO, G. AND RAFFINOT, M. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Experimental Algorithmics (JEA) 5,* 4, http://portal.acm.org.

NAVARRO, G. AND SALMELA, L. 2009. Indexing variable length substrings for exact and approximate matching. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. Springer, 214–221.

NAVARRO, G., SUTINEN, E., AND TARHIO, J. 2005. Indexing text with approximate $q$-grams. *Journal of Discrete Algorithms 3,* 2-4, 157 – 175. Combinatorial Pattern Matching (CPM) Special Issue.

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. *J Mol Biol 48,* 3, 443–453.

NG, K. AND ZUE, V. W. 2000. Subword-based approaches for spoken document retrieval. *Speech Communication 32,* 3, 157 – 186.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9,* 1, 38–71.

NILSSON, S. AND KARLSSON, G. 1999. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications, 17,* 6, 1083–1092.

ORENSTEIN, J. 1982. Multidimensional tries used for associative searching. *Information Processing Letters 14,* 4, 150–157.

OWOLABI, O. 1996. Dictionary organizations for efficient similarity retrieval. *Journal of Systems and Software 34,* 2, 127–132.

OWOLABI, O. AND MCGREGOR, D. R. 1988. Fast approximate string matching. *Software Practice Experience 18,* 4, 387–393.

OZTURK, O. AND FERHATOSMANOGLU, H. 2003. Effective indexing and filtering for similarity search in large biosequence databases. In *BIBE '03: Proceedings of the 3rd IEEE Symposium on BioInformatics and BioEngineering*. IEEE Computer Society, Washington, DC, USA, 359.

PAGH, R. AND RODLER, F. 2001. Cuckoo hashing. In *Algorithms ESA 2001*, F. auf der Heide, Ed. Lecture Notes in Computer Science Series, vol. 2161. Springer Berlin / Heidelberg, 121–133.

PETERSON, J. L. 1980. Computer programs for detecting and correcting spelling errors. *Commun. ACM 23,* 12, 676–687.

PETERSON, J. L. 1986. A note on undetected typing errors. *Communications of the ACM 29,* 7, 633–637.

RAMAN, R., RAMAN, V., AND SATTI, S. R. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms 3,* 4, 43.

RISEMAN, E. M. AND HANSON, A. R. 1974. A contextual postprocessing system for error correction using binary n-grams. *IEEE Trans. Comput. 23,* 5, 480–493.

ROBINSON, J. 1981. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of 1981 ACM SIGMOD international conference on Management of data*. ACM Press, 10–18.

ROUSSOPOULOS, N. AND LEIFKER, D. 1985. Direct spatial search on pictorial databases using packed R-trees. *SIGMOD Rec. 14,* 4, 17–31.

RUSSO, L. AND OLIVEIRA, A. 2005. An efficient algorithm for generating super condensed neighborhoods. *Lecture Notes in Computer Science 3537*, 104–115.

RUSSO, L. M. S., NAVARRO, G., AND OLIVEIRA, A. L. 2008. Fully-compressed suffix trees. In *LATIN 2008: Theoretical Informatics*, E. Laber, C. Bornstein, L. Nogueira, and L. Faria, Eds. Lecture Notes in Computer Science Series, vol. 4957. Springer Berlin / Heidelberg, 362–373.

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

SADAKANE, K. 2007. Compressed suffix trees with full functionality. *Theor. Comp. Sys. 41,* 4, 589–607.

SAKOE, H. AND CHIBA, S. 1971. A dynamic programming approach to continuous speech recognition. In *Proceedings of the Seventh International Congress on Acoustics*. 65–68. paper 20C13.

SAMET, H. 1995. Spatial data structures. In *Modern Database Systems in The Object Model, Interoperability and Beyond*. ACM Press and Addison-Wesley, 361–385.

SAMET, H. 2005. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc.

SANKOFF, D. 2000. The early introduction of dynamic programming into computational biology. *Bioinformatics 16,* 1, 41–47.

SCHEK, H. J. 1978. The reference string indexing method. In *Proceedings of the 2nd Conference of the European Cooperation in Informatics*. Springer-Verlag, 432–459.

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*. ACM Press, 222–229.

SCHUEGRAF, E. J. 1973. Selection of equifrequent work fragments for information retrieval. *Information Storage and Retrieval 9,* 12, 697–711.

SELLERS, P. 1974. On the computation of evolutionary distances. *SIAM Journal of Applied Math 26*, 787–793.

SELLIS, T. K., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R+-Tree: A dynamic index for multi-dimensional objects. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 507–518.

SIEGLER, M., WITBROCK, M. J., SLATTERY, S. T., SEYMORE, K., JONES, R. E., AND HAUPTMANN, A. G. 1997. Experiments in Spoken Document Retrieval at CMU. In *Proceedings of the 6th Text REtrieval Conference*. 291–302.

SKUT, W. 2004. Incremental construction of minimal acyclic sequential transducers from unsorted data. In *COLING '04: Proceedings of the 20th International Conference on Computational Linguistics*. Association for Computational Linguistics, 15.

SUNG, W.-K. 2008. Indexed approximate string matching. In *Encyclopedia of Algorithms*. 408–410.

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

UHLMANN, J. 1991. Satisfying general proximity similarity queries with metric trees. *Information Processing Letters 40*, 175–179.

UKKONEN, E. 1985a. Algorithms for approximate string matching. *Information and Control 64,* 1-3, 100–118.

UKKONEN, E. 1985b. Finding approximate patterns in strings. *Journal of Algorithms 6,* 1, 132–137.

UKKONEN, E. 1993. Approximate string matching over suffix trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, number 684 in Lecture Notes in Computer Science*. Springer-Verlag, 228–242.

UKKONEN, E. 1995. On-line construction of suffix trees. *Algorithmica 14*, 249–260.

VELICHKO, V. AND ZAGORUYKO, N. 1970. Automatic recognition of 200 words. *International Journal of Man-Machine Studies 2,* 3, 223 – 234.

VERONIS, J. 1988. Computerized correction of phonographic errors. *Computers and the Humanities 22,* 1, 43–56.

VIDAL, E. 1986. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letter 4,* 3, 145–147.

VINTSYUK, T. 1968. Speech discrimination by dynamic programming. *Cybernetics 4,* 1, 52–57.

WAGNER, R. A. AND FISCHER, M. J. 1974. The string-to-string correction problem. *Journal of the ACM 21,* 1, 168–173.

WANG, B., XIE, L., AND WANG, G. 2009. A two-tire index structure for approximate string matching with block moves. 197–211.

WEBER, R., SCHEK, H. J., AND BLOTT, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases*. Morgan Kaufmann, 194–205.

WEINER, P. 1973. Linear pattern matching algorithms. In *Proceedings the 14th IEEE Symposium on Switching and Automata Theory*. 1–11.

WILBUR, W. J., KIM, W., AND XIE, N. 2006. Spelling correction in the pubmed search engine. *Information Retrieval 9,* 5, 543–564.

WILLET, P. 1979. Document retrieval experiments using indexing vocabularies of varying size. II. Hashing, truncation, digram and trigram encoding of index terms. *Journal of Documentation 35,* 4, 296–305.

WOODLAND, P., ODELL, J., VALTCHEV, V., AND YOUNG, S. 1994. Large vocabulary continuous speech recognition using HTK. *Acoustics, Speech, and Signal Processing, IEEE International Conference on 3*, 125–128.

WU, S. AND MANBER, U. 1992a. Agrep – a fast approximate matching tool. In *Proceedings of the USENIX Winter Technical Conference*. 153–162.

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

WU, S., MANBER, U., AND MYERS, E. 1996. A subquadratic algorithm for approximate limited expression matching. *Algorithmica 15,* 1, 50–67.

ZALIZNYAK, M. 2006. Telefonica's dispute over an accented domain in Chile. Multilingual Search. Sep 30, 2006.
http://www.multilingual-search.com/telefonica%C2%B4s-dispute-over-an-accented-domain-in-chile/30/09/2006.

ZAMORA, E., POLLOCK, J., AND ZAMORA, A. 1981. The use of trigram analysis for spelling error detection. *Information processing and management 17,* 6, 305–316.

ZEZULA, P., AMATO, G., DOHNAL, V., AND BATKO, M. 2005. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

ZOBEL, J. AND DART, P. 1995. Finding approximate matches in large lexicons. *Software Practice and Experience 25,* 2, 331–345.

ZOBEL, J. AND DART, P. 1996. Phonetic string matching: lessons from information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM Press, 166–172.

ZOBEL, J., MOFFAT, A., AND RAMAMOHANARAO, K. 1998. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst. 23,* 4, 453–490.

ZOBEL, J., MOFFAT, A., AND SACKS-DAVIS, R. 1993. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the 19th international Conference on Very Large Data Bases*. Morgan Kaufmann, 290–301.

ZUE, V., GLASS, J., PHILLIPS, M., AND SENEFF, S. 1989. The MIT SUMMIT speech recognition system: a progress report. In *Proceedings of the workshop on Speech and Natural Language*. HLT '89. Association for Computational Linguistics, Morristown, NJ, USA, 179–189.

## A. PATTERN PARTITIONING PROPERTIES

In this section, we present theorems that provide the theoretical basis for pattern partitioning methods. For the following discussion, we consider the *restricted edit distance* and *restricted edit scripts* that do not contain overlapping edit operations (see Section 2.2 for details).

### A.1. Searching without Transpositions

LEMMA A.1. *Let* $\mathrm{ED}(p, s) \leq k$ *and the string* $p$ *be partitioned into* $j$ *possibly empty contiguous substrings:* $p = p_1 p_2 \ldots p_j$*. Then there is a partition of* $s$ *into* $j$ *possibly empty contiguous substrings:* $s = s_1 s_2 \ldots s_j$*, such that the following inequality holds:*

$$\mathrm{ED}(p_1, s_1) + \mathrm{ED}(p_2, s_2) + \ldots + \mathrm{ED}(p_j, s_j) \leq k \tag{20}$$

*In addition, for every* $p_i$*, its starting position differs from the starting position of* $s_i$ *by at most* $k$.

PROOF. Consider an optimal *restricted* edit script $E$ that transforms $p$ into $s$. The length of $E$ is less than or equal to $k$. Since $E$ does not contain overlapping operations, it is possible to partition $E$ into $j$ non-overlapping scripts $E_1, E_2, \ldots, E_j$ such that each $E_i$ modifies only characters from $p_i$. Insertions between adjacent parts $p_i$ and $p_{i+1}$ that modify no particular $p_i$ can be assigned to either $E_i$ or $E_{i+1}$ arbitrarily. Since $E_i$ do not overlap, their total length is less than or equal to $k$:

$$|E_1| + |E_2| + \ldots + |E_j| = |E| = \mathrm{ED}(p, s) \le k.$$

It can be seen that application of scripts $E_i$ to respective arguments $p_i$ produces substrings $s_i$ that satisfy the condition of the lemma. $\square$

The sum of $j$ non-negative terms $\mathrm{ED}(p_i, s_i)$ in Inequality (20) is at most $k$. Therefore, at least one of terms in Inequality (20) is less than or equal to $\lfloor k/j \rfloor$, which leads to the following theorem:

THEOREM A.2. *[Myers 1994; Baeza-Yates and Navarro 1999]. Let* $\mathrm{ED}(p, s) \le k$ *and the string* $p$ *be partitioned into* $j$ *possibly empty contiguous substrings* $p_1$, $p_2, \ldots$, $p_j$ *starting at positions* $\rho_1$, $\rho_2, \ldots$, $\rho_j$ $(\rho_1 = 1)$. *Then at least one* $p_i$ *matches a substring* $s_{[\tau_1:\tau_2]}$ *with at most* $\lfloor k/j \rfloor$ *errors. In addition, the position of the matching substring satisfies* $|\rho_i - \tau_1| \le k$.

If $j = k+1$, Theorem A.2 has a well-known corollary: if the string $p$ is divided into $k+1$ substrings $p_1, p_2, \ldots, p_{k+1}$, then at least one $p_i$ is an exact substring of $s$ [Wu and Manber 1992b]. The position of the matching substring differs from its original positing in $p$ by at most $k$. This corollary is the basis for partitioning into exact searching.

Another important corollary is obtained directly from Lemma A.1 for $j = 2$, when a string is divided into two parts: $p_1$ and $p_2$. By splitting Inequality (20) into sub-cases $\{\mathrm{ED}(p_1, s_1) \le i,\ \mathrm{ED}(p_2, s_2) \le k-i\}$ for $0 \le i \le k$ and eliminating overlaps, the following theorem is obtained:

THEOREM A.3. *[Mihov and Schulz 2004] Let* $\mathrm{ED}(p, s) \le k$. *If the string* $p$ *is partitioned into possibly empty contiguous substrings* $p_1$ *and* $p_2$, *there exists a partition of the string* $s$ *into* $s_1$ *and* $s_2$ *such that exactly one of the* $k+1$ *inequalities holds:*

$$\begin{array}{lll}
\mathrm{ED}(p_1, s_1) = 0 & \text{and} & \mathrm{ED}(p_2, s_2) \le k \\
\mathrm{ED}(p_1, s_1) = 1 & \text{and} & \mathrm{ED}(p_2, s_2) \le k-1 \\
 & \cdots & \\
\mathrm{ED}(p_1, s_1) = k-1 & \text{and} & \mathrm{ED}(p_2, s_2) \le 1 \\
\mathrm{ED}(p_1, s_1) = k & \text{and} & \mathrm{ED}(p_2, s_2) = 0
\end{array}$$

### A.2. Searching with Transpositions

Now we review well-known extensions of Lemma A.1 and Theorems A.2-A.3 for the case of the Damerau-Levenshtein distance and transposition-aware searching. The main idea behind these extensions consists in "compensation" for *boundary transpositions*.

Let $p = p_1 p_2 \ldots p_j$ be a partition of the string $p$ into $j$ *non-empty* substrings. Given this partition, the $i$-th *boundary transposition* is the edit operation that interchanges the last character of $p_i$ with the first character of $p_{i+1}$. If $i + 2 \le j$ and $p_{i+1} = a$ is the string the length one, then the $i$-th boundary transposition "overlaps" with the $(i+1)$-st boundary transposition, because both of them can modify the character $a$. For the following discussion, we consider only sequences of *non-overlapping boundary transpositions*.

LEMMA A.4. *Let* $\mathrm{ED}(p, s) \le k$ *and the string* $p$ *be partitioned into* $j$ *non-empty contiguous substrings* $p_1, p_2, \ldots, p_j$. *Then there is a sequence of* $0 \le t \le k$ *non-overlapping boundary transpositions that convert* $p_i$ *into* $\bar{p}_i$ *and partitioning of the string* $s$ *into* $j$

*possibly empty contiguous substrings $s_1, s_2, \ldots, s_j$ such that*

$$\mathrm{ED}(\bar{p_1}, s_1) + \mathrm{ED}(\bar{p_2}, s_2) + \ldots + \mathrm{ED}(\bar{p_j}, s_j) \le k - t.$$

*In addition, for every $\bar{p_i}$, its starting position differs from the starting position of $s_i$ by at most $k - t$.*

PROOF. Consider an optimal *restricted* edit script $E$ that transforms $p$ into $s$. According to our assumptions, $E$ contains at most $k$ elementary edit operations of which $0 \le t \le k$ are boundary transpositions. $E$ is a restricted edit script: it does not contain overlapping operations or operations that modify a single substring twice. Therefore, we can decompose $E$ into two non-overlapping scripts $E_1$ and $E_2$ such that $E_1$ includes $t$ boundary transpositions and $E_2$ includes the remaining edit operations. The number of the remaining operations is at most $k-t$. Let $\bar{p} = \bar{p_1}\bar{p_2}\ldots\bar{p_j}$ be the result of applying $E_1$ to $p = p_1 p_2 \ldots p_j$. By construction, $E_2$ transforms $\bar{p}$ into $s$ with at most $k - t$ edit operations. Therefore, $\mathrm{ED}(\bar{p}, s) \le k - t$.

Note that $E_2$ does not contain any boundary transpositions. Therefore, it may contain only the following operations:

— Insertion between substrings $\bar{p_i}$ and $\bar{p}_{i+1}$;
— Insertion, deletion, substitution, or transposition inside substring $\bar{p_i}$.

Using an argument similar to that of Lemma A.1, we decompose $E_2$ into $j$ non-overlapping scripts $\{\bar{E_i}\}$ such that each $\bar{E_i}$ modifies only substring $\bar{p_i}$. Then we apply scripts $\bar{E_i}$ to respective arguments $\bar{p_i}$ and obtain substrings $s_i$ that satisfy

$$\mathrm{ED}(\bar{p_1}, s_1) + \mathrm{ED}(\bar{p_2}, s_2) + \ldots + \mathrm{ED}(\bar{p_j}, s_j) = |E_2|$$

Since $|E_2| \le k - t$, strings $s_i$ satisfy the condition of the lemma. □

The following theorem is an analog of Theorem $A.2$:

THEOREM A.5. *Let $\mathrm{ED}(p, s) \le k$ and the string $p$ be partitioned into $k+1$ contiguous non-empty substrings $p_1$, $p_2$, ..., $p_{k+1}$ starting at positions $\rho_1$, $\rho_2, \ldots$, $\rho_{k+1}$ ($\rho_1 = 1$). Let $u_i$ represent the string obtained from $p_i p_{i+1}$ by the boundary transposition. Let $v_i$ be the string obtained from $u_i$ by removal of the last character, if $|p_{i+1}| > 1$. If $|p_{i+1}| = 1$, we let $v_i = u_i$. Then at least one $p_i$ or $v_i$ matches a substring $s_{[\tau_1:\tau_2]}$ exactly. In addition, the position of the matching substring satisfies $|\rho_i - \tau_1| \le k$.*

PROOF. Consider an optimal *restricted* edit script $E$ that transforms $p$ into $s$. If $E$ does not modify some $p_i$, this $p_i$ satisfies the condition of the theorem. Assume now that $E$ modifies every $p_i$. We use induction by $k$ to show the following: if $E$ contains at most $k$ edit operations and modifies all substrings $p_1$, $p_2$, ..., $p_{k+1}$, there exist $i$ such that $p_i p_{i+1}$ is modified only by boundary transpositions. The first $p_i p_{i+1}$ that has this property also satisfies the condition of the theorem.

*Basis:* If $k = 1$, only a boundary transposition can modify both $p_1$ and $p_2$.

*Inductive step*: We show that if the statement holds for all $k' < k$, it also holds for $k' = k$. The following cases are to be considered:

(1) $p_1 p_2$ is modified only by boundary transpositions.
(2) $p_1 p_2$ is modified only by insertions, deletions, or substitutions. We need at least two edit operations to modify both $p_1$ and $p_2$.
(3) $p_1 p_2$ is modified by the boundary transposition that swaps the last character of $p_1$ and the first character of $p_2$. In addition, $p_1 p_2$ is modified by at least one insertion, deletion, or substitution. Optionally, $p_2$ can be modified by a boundary transposition that swaps the last character of $p_2$ and the first character of $p_3$.

The first case satisfies the inductive condition. Therefore, we have to consider only cases 2 and 3.

Let us decompose $E$ into scripts $E_1$ and $E_2$ such that $E_1$ modifies only $p_1 p_2$ and $E_2$ modifies only $p_3 p_4 \ldots p_{k+1}$. If $E$ does not contain the boundary transposition that swaps the last character of $p_2$ and the first character of $p_3$, $E_1$ has at least two edit operations and $E_2$ contains at most $k-2$ edit operations.

If $E$ contains the boundary transposition that swaps the last character of $p_2$ and the first character of $p_3$, we represent this transposition by two substitutions. The first substitution belongs to the edit script $E_1$ and the second substitution belongs to $E_2$. Then $E_1$ contains at least three edit operations and $E_2$ contain at most $k-2$ edit operations.

In all cases, $E_2$ has at most $k-2$ edit operations and modifies all $k-1$ substrings $p_3, p_4, \ldots, p_{k+1}$. Therefore, by inductive assumption, there exists $i \geq 3$ such that substrings $p_i$ and $p_{i+1}$ are modified only by boundary transpositions. □

Extension of Theorem A.3 follows from Lemma $A.4$:

THEOREM A.6. *Let* $\mathrm{ED}(p, s) \leq k$ *and the string* $p$ *be of the form* $p_1 ab\, p_2$*, where* $a$ *and* $b$ *are single characters. Then at least one of the following assertions is true:*

— *The condition of Theorem* $A.3$ *holds for the string* $p$*, partitioned into* $p_1 a$ *and* $b\, p_2$*, and edit distance* $k$*;*
— *The condition of Theorem* $A.3$ *holds for modified the string* $p_1 ba\, p_2$*, partitioned into* $p_1 b$ *and* $ap_2$*, and edit distance* $k-1$.

## B. HASH FUNCTION FOR ALPHABET REDUCTION

To decrease dimensionality, we use the commonly known method that involves hashing. It consists in projecting the original alphabet $\Sigma$ to an alphabet $\sigma$ of a smaller size using a hash function $h(c)$. In addition to hashing of regular characters, i.e., unigrams, we also consider hashing of the $q$-gram alphabet $\Sigma^q$, which is comprised of all possible $q$-grams.

In our experiments, we use the following hash functions:

— An additive hash function for unigrams: $h(\Sigma_i) = i \bmod |\sigma|$;
— An additive hash function for $q$-grams:

$$h\left(s_{[l:l+q-1]}\right) = \sum_{i=0}^{q-1} \eta^{(q-i)} \mathrm{ASCII}\left(s_{[l+i]}\right) \bmod |\sigma|,$$

where $\mathrm{ASCII}(c)$ is the ASCII value of a character $c$;
— A frequency-optimized hash function for unigrams.

A frequency optimized hash function produces a reduced alphabet, where all characters have approximately equal probabilities of occurrence. It is constructed using frequencies of character occurrence in the dictionary. This data is language-dependent and does not change substantially with $N$: there is no need to recalculate the frequency distribution of alphabet characters and to rebuild the index every time when a new string is added to the dictionary.

The problem of constructing the frequency optimized hash function is hard. Therefore, we opt for an approximate solution based on a simple greedy algorithm. First, the algorithm sorts $\Sigma$ in the order of decreasing character frequencies. Then it uses an iterative procedure that maps characters from the original alphabet to numbers from 1 to $|\sigma|$ in a way that the sum of frequencies of characters mapped to the same number $i$ is approximately equal to $1/|\sigma|$.

## C. DISCUSSION ON SEARCH COMPLEXITY AND MEMORY REQUIREMENTS

### C.1. Assumptions

Upper bounds for retrieval time and index size are guaranteed only on average. They are derived using several simplifying assumptions:

— The number of errors is assumed to be small, i.e., $k = O(1)$;
— The theoretical upper bounds for Mor-Fraenkel-based methods and reduced alphabet generation are derived using a simplified language model, where all characters have equal and independent probabilities of occurrence;
— In the case of the BKT, it is assumed that that probability of encountering a string at distance $i$ from a pivot is the same for every internal node and does not change with $N$.

Additionally, we assume that the following parameters do not change with $N$ either:

— The distribution of patterns;
— The distribution of dictionary strings (strings are generated randomly with replacement);
— The probability of encountering a positional $q$-gram;
— The probability of encountering a string with a given frequency vector.

Note that in the case of sequence-based filtering methods, it is easy to prove that retrieval time has a linear term $\beta(n,k)N$, but it is difficult to obtain an analytical expression for $\beta(n,k)$. Similarly, we empirically verify that the number of frequency vectors (or reduced alphabet strings) $M(N)$ is small in comparison to $N$, but do not provide an analytical justification.

### C.2. Prefix Trees

Prefix trees (the string trie, the FB-trie, and the $k$-errata tree) are discussed in Section 6.1, p. 20.

*C.2.1. Search Complexity: String Trie.* For simplicity, we consider only transposition-unaware searching. Given a *complete* string trie, the number of nodes visited by the search procedure can be upper bounded by the total number of $k$-neighborhoods of various search pattern prefixes:

$$\sum_{i=1}^{n} U_k(p_{[1:i]})$$

Since $U_k(p_{[1:i]}) \leq \frac{12}{5}(i+1)^k(|\Sigma|+1)^k$ [Ukkonen 1993], the number of visited nodes is upper bounded by:

$$\sum_{i=1}^{n} \frac{12}{5}(i+1)^k(|\Sigma|+1)^k = O(n^{k+1}|\Sigma|^k) \tag{21}$$

This estimate also holds for path-compressed tries, which are lossless compact representations of complete tries.

Let the node $\xi$ spell the string $s$. The time spent in $\xi$ depends on the choice of algorithm that verifies whether $\text{ED}(p, su) \leq k$ for some string $u$. This verification can be accomplished in $O(k)$ time by computing $2k+1$ main diagonals of DP matrix or in $O(1)$ time using the universal Levenshtein automaton [Wu et al. 1996; Mihov and Schulz 2004]. If $k = O(1)$, a search query can be satisfied in $O(|\Sigma|^k n^{k+1})$ time.

Navarro and Baeza-Yates [2000] presented an average case analysis of suffix trees that is also valid for generic string tries. Below we explain why their result is only marginally better than Expression (21) in many practical situations including our experimental setup.

According to their analysis, the number of visited nodes is upper bounded by

$$\left\{ |\Sigma|^{\frac{k}{\sqrt{|\Sigma|}/c-1}} + (1+k/n)^{2(n+k)}(n/k)^{2k} \right\} |\Sigma|^k, \tag{22}$$

where $c < e$ and $|\Sigma| > c^2$. If $|\Sigma| \leq c^2$, the estimate (22) does not contain the first summand. According to empirical assessment by Navarro and Baeza-Yates [2000], $c$ should be close to 1, but this has not been demonstrated analytically.

In the case of natural languages $\sqrt{|\Sigma|} \geq c \geq e$. Therefore, the first summand inside the curly brackets in (22) can be lower bounded by $|\Sigma|^{\left(\sqrt{|\Sigma|}/e-1\right)^{-1}}$, which is a decreasing function of variable $|\Sigma|$ for $\sqrt{|\Sigma|} > e$. For $|\Sigma| \leq 36$, this lower bound is greater than $19$. Thus, for our datasets Expression (22) is greater than

$$|\Sigma|^k 19^k \tag{23}$$

For the DNA data used in our experiments, $n \geq 11$, $k \leq 3$, and $k \leq \sqrt{n}$. Therefore, the second summand inside the curly brackets in (22) is at least $n^k$ and the Expression (22) is greater than

$$|\Sigma|^k n^k \tag{24}$$

It can be seen that neither (23) in the case of natural language data nor (24) in the case $k \leq \sqrt{n}$ do not significantly improve over Expression (21). Furthermore, they predict a much faster growth (depending on $k$) than that observed in our experiments, where retrieval time increases approximately as $10^k$ (see Section 8.3.2, p. 54).

*C.2.2. Index size.* In the next subsection we present Theorem C.1, which estimates the *index overhead* of the path-compressed trie, i.e., the difference between the size of the trie and the size of the dictionary. According to Theorem C.1, p. 81, is upper bounded by:

$$(2P+2)N,$$

where $P$ is the size of the pointer.

If $\lambda = 10$ (the average string length) and $P = 4$, from Theorem C.1 it follows that the size of the trie is at most twice as large as the size of the dictionary itself. In practice (see Table IX, p. 65), the size of the trie varies from 115 to 170 percent of the dictionary size. It can be also seen that the FB-trie uses about twice as much space as the regular trie.

*C.2.3. Upper Bounds for Size of Path-Compressed Trie.* The path-compressed trie has two components: a set of edge labels that represent dictionary strings and auxiliary data that supports the structure of a hierarchical tree. The index overhead is the difference between the size of the path-compressed trie and the total size of dictionary strings. It is convenient to estimate overhead associated with a subtree divided by the number of strings stored in this subtree. Because each subtree is associated with a string prefix (shared among all strings in this subtree), we refer to this overhead as the “per-prefix” overhead.

For the purposes of this analysis, we assume the following storage model: an internal node $\xi$ stores a string label of the incoming edge (connecting $\xi$ with its parent node), a counter of child nodes, pointers to child nodes, and lengths of string labels for outgoing edges (connecting $\xi$ with its child nodes). A leaf node stores a counter of child nodes that is equal to zero and the string label of the incoming edge. The dictionary is represented by concatenated strings, each of whom is terminated with a special zero-code character. In addition, we assume that both the alphabet size and string lengths are less than or equal to 255. Thus, all counters need one byte. The size of the pointer is $P$ bytes.

THEOREM C.1. *Given the storage model described above, the index overhead of the path-compressed string trie does not exceed $N(2P+2)$, where $N$ is the number of dictionary strings.*

PROOF. The proof consists in recursive evaluation of the per-prefix overhead. Assume that the node $\xi$ is at the level $n$ and its per-prefix overhead is $E_n$. Let the length of the incoming label be $l_p$ and the number of children be $n_c$. Two cases have to be considered.

*$\xi$ is an internal node.* Because the trie is path-compressed, $n_c \geq 2$. To store the pointers to child nodes, the counter of child nodes, and lengths for outgoing edge labels we need $n_c + 1 + n_c P = n_c(P+1) + 1$ bytes. The path-compressed trie keeps only one copy of the incoming edge label, instead of $n_c$ copies that would have been kept if the dictionary were represented by concatenated strings. Therefore, storing data in the form of the trie decreases the per-prefix overhead by $l_p(n_c - 1)$ bytes at the level $n+1$.

To obtain the per-prefix overhead associated with a child of $\xi$, we first sum the per-prefix overhead $E_n$ (associated with $\xi$) and the overhead incurred by storing auxiliary data (pointers and counters). Then we subtract savings achieved by sharing the incoming edge label among children of $\xi$ and divide the result by the number of child nodes $n_c$:

$$f(l_p, n_c) = \left(n_c(P+1) + 1 + E_n - (n_c - 1)l_p\right)/n_c = P + 1 + (E_n + 1)/n_c - (1 - 1/n_c)l_p$$

For $l_p \geq 1$ and $n_c \geq 2$ the function $f(l_p, n_c)$ has negative partial derivatives with respect to $l_p$ and $n_c$. Therefore, it achieves the maximum at $l_p = 1$ and $n_c = 2$. This implies

$$E_{n+1} \leq P + 3/2 + E_n/2 - 1/2 = P + 1 + E_n/2$$

From $E_0 = 0$ it follows that

$$E_n \leq (P+1)\left(1 + 1/2 + \ldots + 1/2^{n-1}\right) \leq 2P + 2$$

*$\xi$ is a leaf.* The node stores only a string label of incoming edge and a one-byte counter. The string label corresponds to the suffix of a dictionary string and does not incur any overhead. Because each string in the dictionary is terminated with a zero-code character that uses the same space as the counter (one byte), the overhead of $\xi$ is equal to $E_n$. Above it has been demonstrated that $E_n \leq 2P + 2$. □

## C.3. Neighborhood Generation

Neighborhood generation methods are discussed in Section 6.2, p. 25.

*C.3.1. Search Complexity: Full Neighborhood Generation.* Neighborhood generation reduces approximate searching to exact dictionary searching by building a string neighborhood. Let $n$ be the length of the search pattern $p$. Both the size of the full $k$-neighborhood and its construction time are $O(n^k|\Sigma|^k)$ (see Equation (12), p. 26). Since exact dictionary searching can be done in $O(n)$ time, the average search time of full neighborhood generation is $O\left(n^{k+1}|\Sigma|^k\right)$.

*C.3.2. Search Complexity: Reduced Alphabet Neighborhood Generation*

THEOREM C.2. *The average retrieval time of reduced alphabet generation is:*

$$O\left(n^{k+1}|\sigma|^k \left\{1 + \frac{L(n,k)}{|\sigma|^n}\right\}\right) \tag{25}$$

PROOF. During the search time, the pattern $p$ is converted into the reduced pattern $p' = h(p)$ by a character-wise application of the hash function $h(c)$. Then the search procedure recursively creates the full neighborhood of $p'$ using the reduced alphabet $\sigma$ and a wildcard neighborhood of the original pattern. Because the size of the wildcard neighborhood is much smaller than a size of the full neighborhood, the computational cost of this step is $O(n^k|\sigma|^k)$.

For each string $s'$ from the full neighborhood of $p'$, the search procedure retrieves dictionary strings $s_i$ such that $h(s_i) = s'$ and verifies whether $s_i$ matches $p$ by comparing $s_i$ to elements from the wildcard neighborhood of $p$. The total number of comparison operations at this step is equal to $\mu|s'|$, where $\mu$ is the number of $s_i$.

We model $\mu$ as a number of outcomes in a series of Bernoulli trials. Each trial consists in picking up a random string $s_i$ of the length $n' = |s'|$ and verifying whether $h(s_i) = s_i$. In addition, we assume that the hash function $h(c)$ performs a uniform mapping and, therefore, the probability of success is equal to $|\sigma|^{-n'}$. Under these assumptions, the expected number of operations (equal to $\mu n'$) is upper bounded by

$$\frac{N(n')n'}{|\sigma|^{n'}} \leq \frac{L(n,k)n'}{|\sigma|^{n'}}, \tag{26}$$

where $N(n')$ is the number of dictionary strings with length $n'$ and $L(n,k) = \max_{|i-l|\leq k} N(i)$.

To obtain an upper bound for the average number of comparisons associated with the full neighborhood of $p'$, we will sum up the values of Expression (26) for all neighborhood strings $s'$. Because for $|\Sigma| \geq 2$ and $n' \geq 1$, $n'|\sigma|^{-n'}$ is a monotonically decreasing function of $n'$, for $n' > n$ each summand is upper bounded with $L(n,k)n|\sigma|^{-n}$ It remains to estimate the summands for $n' \leq n$.

If $n'$ (the length of $s'$) is equal to $n - t$, then $s'$ is obtained from $p'$ using $t$ deletions and $k - t$ other edit operations. Because the number of ways to apply $t$ deletions to a string of the length $n$ is $O(n^t)$, the number of neighborhood strings of the length $n - t$ is upper bounded by

$$O(n^t \times n^{k-t}|\sigma|^{k-t}) = O(n^k|\sigma|^{k-t}). \tag{27}$$

The number of neighborhood strings that contain $n$ or more characters is upper bounded by the size of the full reduced-alphabet neighborhood: $O(n^k|\sigma|^k)$.

Combining these observations and Expression (26), we get that the expected number of comparisons is upper bounded by

$$\sum_{t=0}^{k} \frac{L(n,k) \times (n-t)n^k|\sigma|^{k-t}}{|\sigma|^{n-t}} < \frac{kn^{k+1}L(n,k)}{|\sigma|^{n-k}}. \tag{28}$$

The upper bound for the total number of operations is obtained from (28) by adding the number of operations required to generate the full reduced-alphabet neighborhood of $p'$, and to search for its each elements exactly. □

*C.3.3. Search Complexity: Mor-Fraenkel Method.* Let us consider the transposition-unaware modification of the Mor-Fraenkel method for $k = 1$. Given the pattern $p$, the search procedure generates $n$ triples $(\Delta_i(p), i, p_{[i]})$. For each triple $(\Delta_i(p), i, p_{[i]})$, we retrieve dictionary triples $(s', j, a)$ such that $\Delta_i(p) = s'$.

The time needed to retrieve triples using $n$ keys $\{\Delta_i(p)\}$ is $O(n^2)$. The time needed to process retrieved triples is proportional to the number of strings $s \in W$ that satisfy $\Delta_i(p) = \Delta_j(s)$ for some $i$ and $j$. In the worst case, the number of such strings is $O(n^2|\Sigma|)$, but typically few of those belong to the dictionary.

THEOREM C.3. *The average retrieval time of the Mor-Fraenkel method is:*

$$O\left(n^{k+1}\left\{1+\frac{n^{k-1}L(n,k)}{|\Sigma|^{n-k}}\right\}\right). \tag{29}$$

PROOF. In the general case, the search procedure generates $O(n^k)$ strings by applying up to $k$ deletions and searches for each generated string in an $l$-deletion index ($l \leq k$). Each query to the $l$-deletion index runs in $O(n)$ time and returns at most $O(n^k|\Sigma|^k)$ strings. We model the actual number of returned strings as a total number of successes in $O((k+1)^2 n^k)$ series of Bernoulli trials.

Let $p'$ be obtained from $p$ by deleting $k_1 \leq k$ characters, which can be done in $O(n^{k_1})$ ways. Each $0 \leq k_2 \leq k$ corresponds to a series of trials, where we check whether $O(n^{k_2}|\Sigma|^{k_2})$ strings obtainable from $p'$ by $k_2$ insertions belong to the dictionary. Because all such strings have length $n-k_1+k_2$, we assume that this verification succeeds with a probability equal to $L(n,k)|\Sigma|^{-n+k_1-k_2}$, where $L(n,k)=\max_{|i-l|\leq k} N(i)$ and $N(i)$ is the number of dictionary strings of the length $i$. Therefore, the expected number of successes in a single series of trials is

$$O\left(\frac{n^{k_1+k_2}L(n,k)}{|\Sigma|^{n-k_1}}\right) = O\left(\frac{n^{2k}L(n,k)}{|\Sigma|^{n-k}}\right) \tag{30}$$

By adding time spent to perform $O(n^k)$ exact searches, from (30) and $k=O(1)$ we obtain the following asymptotic upper bound for the average retrieval time of the Mor-Fraenkel method:

$$O\left(n^{k+1}\left\{1+\frac{n^{k-1}L(n,k)}{|\Sigma|^{n-k}}\right\}\right).$$

Because $k=O(1)$, correction for transpositions takes little additional time and, therefore, the same upper bound holds for the transposition-aware version. □

*C.3.4. Search Complexity: Hybrid Neighborhood Generation.* Hybrid neighborhood generation combines computation of the full $(k-l)$-neighborhood at query time with the Mor-Fraenkel method that evaluates and stores deletion-only $t$-neighborhoods during indexing ($t \leq l$). At query time, the search procedure generates the full $(k-l)$-neighborhood of the pattern $p$. The neighborhood contains $O((n|\Sigma|)^{k-l})$ elements. Each element of this neighborhood is then searched for in the index of the Mor-Fraenkel method within $l$ errors. Retrieval time depends on the length of the searched string, which ranges from $n-k+l$ to $n+k-l$.

Analogous to the analysis of reduced alphabet neighborhood generation, we upper bound the contribution of strings longer than $n$ characters by a larger term (that represents retrieval time for the pattern exactly $n$ characters long). Then, using the upper bound for the number of neighborhood strings shorter than $n$ given by Equation (27) and the estimate for the Mor-Fraenkel method given by Equation (29) for $k=l$, we obtain that for $k \geq l$ the total average retrieval time is upper bounded by

$$\sum_{t=0}^{k-l} O\left((n-t)^{l+1}\left\{1+\frac{(n-t)^{l-1}L(n,l)}{|\Sigma|^{n-t-l}}\right\}\times n^{k-l}|\Sigma|^{k-t-l}\right) =$$

$$O\left(n^{k+1}|\Sigma|^{k-l}\left\{1+\frac{n^{l-1}L(n,l)}{|\Sigma|^{n-l}}\right\}\right)$$

For $k \leq l$, the algorithm works as the Mor-Fraenkel method. Therefore, in the general case the average retrieval time is upper bounded by

$$O\left(n^{k+1}|\Sigma|^{k-k'}\left\{1+\frac{n^{k'-1}L(n,k')}{|\Sigma|^{n-k'}}\right\}\right), \text{ where } k'=\min(k,l). \quad (31)$$

*C.3.5. Search Complexity: Dense Dictionaries and Short Patterns.* Reduced alphabet generation, the Mor-Fraenkel method, and hybrid neighborhood generation are filtering methods. The average time of their verification steps depend on $L(n,k)$, which is approximately equal to the number of strings of the length $n$. In the upper bounds given by Expressions (25), (29), and (31) the cost of verification is represented by a second summand inside the curly brackets.

If the dictionary is sparse and $n$ is sufficiently large, this summand is less than one. If the dictionary is dense, $L(n,k) \approx |\Sigma|^n$. Consequently, reduced alphabet generation becomes equivalent to full neighborhood generation, while for $k > 1$ the Mor-Fraenkel method becomes slower than full neighborhood generation (by the factor of $n^{k-1}$). We suppose that a transducer-based variant of the Mor-Fraenkel method would be slower than full neighborhood generation by the factor of $2^k$, but we have not proved it rigorously.

A similar performance degradation is observed for short patterns and prefix-dense dictionaries. Even though short patterns are infrequent and, thus, contribute to the overall average retrieval time with a smaller weight, retrieval of short strings can be quite expensive. In our experiments, we have found that full neighborhood generation is significantly more efficient than hybrid neighborhood generation-2 for $k = 3$ and small $n \approx 4$.

*C.3.6. Index size: Full and Reduced Alphabet Generation.* One of the fastest methods for exact searching is hashing based an open addressing. It comprises a dictionary stored as a plain file and a hash table that is used to locate dictionary strings. The hash table represents the index overhead and occupies $O(N)$ space, where implied constant largely depends on an implementation (a similar estimate is obtained if the dictionary is stored as a trie, see Theorem C.1, Section C.2.2).

A straightforward implementation of a hash table keeps $N/\alpha$ pointers, where $\alpha$ is a load factor of the hash table. A space-efficient implementation described in 4.1 uses only several extra bits per string.

In the case of reduced alphabet generation, the overhead is proportional to the number of unique "hashed" strings. This number is typically much smaller than $N$. Therefore, reduced alphabet generation has a smaller index overhead.

*C.3.7. Index Size: Super-Condensed Neighborhood.* Our implementation of super-condensed neighborhood generation employs an inverted file for a subset of dictionary substrings. This solution is fast but space-inefficient. A more space-efficient solution would employ a suffix-tree-like data structure that uses space $O(\lambda N)$. Further reduction in space may be achieved by using a compressed suffix tree/array.

*C.3.8. Index Size: Mor-Fraenkel method and Hybrid Neighborhood Generation.* The indexing step of the Mor-Fraenkel method consists in building and storing deletion-only $l$-neighborhoods of dictionary strings for all $l \leq k$. The number of neighborhood elements can be upper bounded by $O((\lambda_m)^k N)$, where $\lambda_m$ represents the maximum length of dictionary strings.

Each neighborhood element requires $O(\lambda_m)$ byte of storage. Therefore, the index size can be upper bounded by $O((\lambda_m)^{k+1} \cdot N)$. Hybrid neighborhood generation method, stores deletion-only $t$-neighborhoods for $t \leq l < k$. Therefore, both the index size and the index overhead are $O((\lambda_m)^{l+1} \cdot N)$.

A transducer-based variant of the Mor-Fraenkel method has smaller space requirements, but we are unaware of analytical estimates for $k > 1$.

## C.4. Metric Space Methods

Metric space methods are discussed in Section 6.3, p. 34.

*C.4.1. Search complexity.* Running time of the BKT is known to be $O(N^{\beta(k)})$ on average, where $\beta(k) < 1$ [Baeza-Yates and Navarro 1998]. According to the evaluation of Baeza-Yates and Navarro [1998], $\beta(1) \approx 0.69$ and $\beta(2) \approx 0.87$ if $b = 20$.

*C.4.2. Index size.* Each BKT leaf node represents a bucket that stores at most $b$ elements. The overhead associated with the BKT index is equal to the space occupied by internal nodes. To calculate this overhead, we estimate an expected number of internal nodes using a recursive analysis similar to that of Baeza-Yates et al. [1994]. To this end, we model the random process of distributing the strings among subtrees during indexing using a multinomial distribution.

Henceforth $I(N)$ denotes the number of internal nodes.

PROPERTY C.4. *It can be seen that $I(N)$ satisfies the following:*

— *$I(N) \leq N - 1$, because each internal node stores a (pivot) string and at least one of the strings will be stored in a bucket;*
— *If $N \leq b$ then $I(N) = 0$, because all elements are stored in a single BKT bucket.*

THEOREM C.5. *The average index overhead of the BKT can be asymptotically upper bounded by:*

$$O(\lambda_m P \min(N, (N/b)^{\gamma})), \text{ for some } \gamma > 0. \tag{32}$$

PROOF. Consider one step of the indexing process that consists in choosing a random pivot and dividing the remaining $N - 1$ strings into $\lambda_m$ subsets $\{W_i\}$ ($\lambda_m$ is the maximum length of dictionary strings). Recall that $W_i$ contains strings at distance $i$ from the pivot. Let $B_i$ denote an event of placing a string into the $i$-th subset on a given trial. According to our assumptions, probabilities $\Pr(B_i)$ do not depend neither on the choice of the pivot nor on $N$.

Let $\epsilon$ be a small positive number and $A_i$ be an event such that, after $N-1$ trials, the actual number of strings $j_i$ in the $i$-th subset $W_i$ diverges from the expected number of strings by more than $\epsilon(N - 1)$. Note that the number of strings in $W_i$ follows the binomial distribution with the probability $\Pr(B_i)$. Therefore, $|j_i - (N - 1)\Pr(B_i)| > \epsilon(N - 1)$ in the case of the event $A_i$. Let also $A = \cup_{i=1}^{\lambda_m} A_i$ be an event such that the number of strings in at least one of the subsets diverges from the respective expected value by more than $\epsilon(N - 1)$.

According to the Hoeffding's inequality [Hoeffding 1963], $\Pr(A_i) \leq 2e^{-2\epsilon^2(N-1)}$ and $\Pr(A)$ satisfies the following inequality:

$$\Pr(A) = \Pr\left(\bigcup_{i=1}^{\lambda_m} A_i\right) \leq \sum_{i=1}^{\lambda_m} \Pr(A_i) \leq 2\lambda_m e^{-2\epsilon^2(N-1)} \tag{33}$$

In what follows, we estimate the expected value of $I(N)$ using expectations conditioned on the event $A$ and its complement $\bar{A}$:

$$E(I(N)) = E(I(N)|A)\Pr(A) + E(I(N)|\bar{A})\Pr(\bar{A}) \tag{34}$$

In the case of the event $A$, we pessimistically assume that the number of internal nodes is equal to $N-1$. Then, from Inequality (33) it follows that

$$E(I(N)|A)\mathbf{Pr}(A) \le (N-1)\mathbf{Pr}(A) \le 2\lambda_m(N-1)e^{-2\epsilon^2(N-1)}$$

For $\alpha > 0$, $xe^{-\alpha x}$ achieves the maximum value of $\frac{1}{\alpha e}$. By setting $\alpha = 2\epsilon^2$, $x = N-1$, and applying $xe^{-\alpha x} \le \frac{1}{\alpha e}$ to the inequality above, we obtain the following upper bound:

$$E(I(N)|A)\mathbf{Pr}(A) \le \frac{\lambda_m}{\epsilon^2 e} \tag{35}$$

$E(I(N)|\bar{A})$ is expressed using the sum of expected numbers of internal nodes associated with subsets $\{W_i\}$. This sum should be incremented by one to account for the internal node that stores the current pivot:

$$E(I(N)|\bar{A}) = 1 + \sum_{i=1}^{\lambda_m} E(I(j_i|)), \tag{36}$$

where $j_i$ is the actual number of strings in $W_i$.

In the case of the event $\bar{A}$, every $j_i$ satisfies $|j_i - (N-1)\mathbf{Pr}(B_i)| \le \epsilon(N-1)$. Therefore, Equation (36) implies the following inequality:

$$E(I(N)|\bar{A}) \le 1 + \sum_{i=1}^{\lambda_m} \max_{|j_i-(N-1)\mathbf{Pr}(B_i)|\le\epsilon(N-1)} E(I(j_i)) \tag{37}$$

By upper bounding $\Pr(\bar{A})$ with 1, combining Property C.4, Equation (34), Inequalities (35), and (37), we obtain that the expected number of internal nodes satisfies the following recurrent inequality:

$$E(I(N)) \le C(\epsilon) + \sum_{i=1}^{\lambda_m} \max_{|j_i-(N-1)\mathbf{Pr}(B_i)|\le\epsilon(N-1)} E(I(j_i)),\ \text{where } C(\epsilon) = 1 + \frac{\lambda_m}{\epsilon^2 e}$$

$$E(I(N)) = 0,\ N \le b \tag{38}$$

One approach to cracking Recurrence (38) is to assume that there exist a solution that is monotonic with respect to $N$. This assumption allows one to simplify (38). Then, the estimate for $E(I(N))$ can be found using the simplified recurrence. If the function obtained is monotonic, then it is also a solution for the original recurrence. The validity of this approach follows from the monotonicity of the right-hand side of Recurrence (38) with respect to $E(I(j_i))$ and can be proved using induction.

This implies that a solution should be evaluated using the following recurrence:

$$E(I(N)) = C(\epsilon) + \lambda_m E(I(\lfloor (N-1)(\rho+\epsilon)\rfloor)), \tag{39}$$

where $\rho = \max \mathbf{Pr}(B_i)$ is the maximum probability of placing a string into $W_i$. If $\epsilon < 1-\rho$ and, consequently $\rho + \epsilon < 1$, we can unfold Recursion (39) to obtain the following asymptotic upper bound:

$$E(I(N)) = O\left(C(\epsilon)\lambda_m^{\log_{\frac{1}{\rho+\epsilon}} N/b}\right) = O\left(\left(1+\frac{\lambda_m}{\epsilon^2 e}\right)(N/b)^{\log_{\frac{1}{\rho+\epsilon}} \lambda_m}\right) = O((N/b)^\gamma). \tag{40}$$

If one of the probabilities $\mathbf{Pr}(B_i)$ and, consequently, their maximum $\rho$ are close to one, a majority of strings will be placed into the $i$-th subtrees, which will result in a highly unbalanced tree. This tree will have large height and a lot of internal nodes. This intuition is supported by Expression (40), which converges to infinity when $\rho$ approaches $1-\epsilon$.

Because $\text{ED}(u, v) \le \lambda_m$ for any $u, v \in W$, each internal node contains at most $\lambda_m$ child pointers of size $P$. Thus, the average overhead is upper bounded by

$$O(\lambda_m P(N/b)^\gamma), \text{ for some } \gamma > 0.$$

Combining this estimate with inequality $E(I(N)) \le N$, we obtain the final asymptotic upper bound for the average index overhead. □

For small $b$ the index overhead can be larger than the dictionary. As $b$ grows, the overhead decreases. For example, for $b = 200$, which yields an optimal retrieval speed in the case of natural language data, the overhead is 4 percent of the dictionary size (see Table IX, p. 65).

## C.5. Pattern Partitioning

*C.5.1. Search complexity: General Observations.* Pattern partitioning methods are discussed in Section 7.1, p. 37. In what follows we show that the retrieval time of the inverted $q$-gram file (see Section 7.1.2, p. 38), which employs partitioning into exact searching, is proportional to the number of dictionary strings $N$.

In general, the time spent on checking candidate strings is proportional to probabilities of encountering string fragments, which, according to our assumptions, do not change with $N$. If string fragments are small, the overall retrieval time is dominated by the checking step. In this case, it is possible to show that the retrieval time of every filtering method that relies on indexing of small string fragments is proportional to $N$.

*C.5.2. Search complexity: Inverted $q$-gram File.* The first assumption ensures that the probability of encountering any given positional $q$-gram does not depend on $N$, but allows for repeating strings. It also guarantees that given the set of positional $q$-grams $\{(s_i, \tau_i)\}$, the probability of encountering a string that includes $q$-grams $s_1, s_2, \ldots$ at positions $\tau_1, \tau_2, \ldots$ (i.e., the probability of $q$-gram co-occurrence), does not depend on $N$.

Let $\Pr(p)$ be the probability of encountering search pattern $p$. Let $C$ denote the computational complexity of the search procedure of the inverted $q$-gram file. Then the expected value of $C$ is

$$E(C) = \sum_i E(C|p_i)\Pr(p_i), \tag{41}$$

where the summation is carried over all possible patterns. $E(C|p_i)$ is the conditional expected value of the number of computations. It can be represented as a sum of three terms:

$$E(C|p_i) = E_{\text{filt}}(p, k) + E_{\text{check}}(p, k) + E_{\text{aux}}(p, k),$$

where $E_{\text{filt}}(p, k)$ is the expected value of the computational cost of the filtering step, $E_{\text{check}}(p, k)$ is the expected value of the computational cost of the checking step, and $E_{\text{aux}}(p, k)$ is the expected value of the computational cost associated with search in the auxiliary index. As we explain below the expected value satisfies:

$$0 < \beta_1(p, k)N \le E(C|p_i) \le \beta_2(p, k)N, \tag{42}$$

where functions $\beta_1(p, k)$ and $\beta_2(p, k)$ depend only on the search pattern $p$ and the maximum allowed distance $k$. By plugging Inequality (42) into Equation (41), we obtain that the overall expected computational cost is $O(\beta(k)N)$ for some $\beta(k)$ that depends only on $k$.

Assuming that the auxiliary index is represented by a plain file (that keeps strings of limited length and is searched sequentially) and string distribution does not depend on $N$, it is easy to show that $E_{\text{aux}}(p, k) \le \beta_{\text{aux}}(p, k)N$. In what follows, we show that the expected computational cost of the filtering step is $\beta_{\text{filt}}(p,k)N$. Using a similar argument and the assumption that the probability of co-occurrence of positional $q$-grams does not depend on $N$, it is possible to show that an analogous estimate holds for the computational cost of the checking step.

A search algorithm that employs the inverted $q$-gram index can be outlined as follows:

(1) Given the pattern $p$ and the maximum allowed distance $k$, the search procedure computes the set of positional $q$-grams $\{(s_i,\tau_i)\}$ that is determined only by $p$ and $k$;
(2) Then it retrieves sorted inverted lists $S_i$ of positional $q$-grams $\{(s_i,\tau_i)\}$ computed in Step 1;
(3) A new set of inverted lists is obtained by merging some of the inverted lists computed in Step 2;
(4) A new set of inverted lists is obtained by intersecting some of the inverted lists computed in Step 3;
(5) Finally, inverted lists computed in Step 4 are merged together.

Note that an intersection and/or a union of sorted inverted lists $S_1$ and $S_2$ can be computed by the merging algorithm in time proportional to $|S_1|+|S_2|$ (in some cases it can be done faster, e.g., when one list is much longer than another). Therefore, the computation of intersections and/or unions in each of the Steps 3 to 5 can be done in time proportional to the total number of inverted list elements, produced in a previous step.

Because the total number of inverted list elements does not increase through Steps 3 to 5, the running time of the filtering step is upper-bounded by $\alpha \times 3 \times \sum |S_i|$, for some $\alpha > 0$. On the other hand, the running time is lower bounded by the total length of inverted lists retrieved in Step 2, which is equal to $\sum |S_i|$.

The inverted list $S_i$ corresponds to the positional $q$-gram $\{(s_i,\tau_i)\}$. Let $\Pr(S_i) = \Pr(s_i,\tau_i)$ denote the probability of $q$-gram $s_i$ to occur at position $\tau_i$. Then the expected value of $\sum |S_i|$ is:

$$\sum_i N\mathbf{Pr}(S_i) = N\sum_i \mathbf{Pr}(s_i,\tau_i) = \beta_{\text{filt}}(p,k)\cdot N,$$

where the factor $\beta_{\text{filt}}(p,k)$ depends only on the pattern $p$ and the maximum allowed edit distance $k$.

*C.5.3. Index size: Inverted $q$-gram File.* The inverted $q$-gram file has four components: a list of dictionary strings, an indexed list of $q$-grams with pointers to corresponding inverted lists, inverted lists, and an auxiliary index built over short strings. Dictionary strings can be stored as a plain file, which uses $O(\lambda N)$ bytes, where $\lambda$ is the average length of dictionary string. Because there are at most $|\Sigma|^q$ non-positional $q$-grams and at most $\lambda_m|\Sigma|^q$ positional $q$-grams, the space requirement for the list of indexed $q$-grams is $O(\lambda_m|\Sigma|^q)$. The auxiliary index uses $O(\beta(q,k)N)$ space, where $\beta(q,k) \ll 1$ depends only on $q$ and $k$.

The size of the third index component is proportional to the total number of inverted lists entries. Each positional $q$-gram corresponds to a unique inverted list entry, while each padded dictionary string of the length $m$ contains $m$ positional $q$-grams. Therefore, the total number of inverted list entries is $\lambda N$ and uncompressed inverted lists use space $\lambda NP$, where $P$ is the size of the pointer. The total size of the index is $\lambda NP + O(\lambda N) + \lambda_m|\Sigma|^q + \beta(q,k)N = O(\lambda_m|\Sigma|^q + \lambda N(P+1))$. The index overhead, i.e., the difference between the size of the index and the size of the dictionary, is $O(\lambda_m|\Sigma|^q + \lambda NP)$.

If $P = 4$, the size of uncompressed inverted $q$-gram file is at least five times the size of the dictionary. To reduce the index size, we use a well-known approach that consists in compression of inverted lists using variable-length codes [D'Amore and Mah 1985; Zobel et al. 1993]. An inverted list is essentially a list of integer numbers. Therefore, it can be sorted and represented as a sequence of differences between adjacent elements. Since most differences are small, they can be stored using little space.

Consider, for example, an integer list that contains numbers 355, 100, 457, 657. The compression algorithm first re-sorts the list and obtains the list $x = (100, 355, 457, 657)$. Then all but the first element of $x$ are substituted with $x_i - x_{i-1}$. As a result, $x$ is transformed into the list $(100, 255, 102, 200)$, where each list element can be stored using one byte. In the general case, the differences may not fit into one byte and have to be encoded using codes of variable length: the smaller is the number, the shorter is the code. In our implementation, we use the variable-byte code, which is aligned by byte boundaries (see, e.g., [Scholer et al. 2002] for details).

If compression is applied, the $q$-gram index overhead is $O(\lambda_m|\Sigma|^q + \alpha_{\text{comp}} \cdot \lambda NP)$, where compression ratio $\alpha_{\text{comp}}$ depends on the choice of compression algorithm, type of $q$-gram and its size $q$. In the case of non-positional $q$-grams and $q = 2$, $\alpha_{\text{comp}}$ can be less than $0.2$ [Zobel and Dart 1995]. In our experiments, we index *positional* $q$-grams longer than two characters and achieve $\alpha_{\text{comp}}$ in the range from 0.4 to 0.8.

The reason for this difference is that the number of occurrences of a positional $q$-gram with $q \geq 3$ is usually much smaller than that of a non-positional $q$-gram with $q = 2$. Therefore, a typical difference between adjacent inverted list elements is much larger and requires many more bits to be encoded. Division of a dictionary based on string lengths leads to even larger differences between adjacent inverted list elements and to less space-efficient encoding. Also note that we use byte aligned codes, which are less space efficient than bit-aligned codes.

### C.6. Vector Space Frequency Distance Methods

Mapping to vector methods are discussed in Section 7.2, p. 42.

*C.6.1. Search complexity.* According to our assumptions, for any given frequency vector $x$, the probability of encountering the string $s$ such that $\text{vect}(q\text{-grams}(s)) = x$ does not depend on the size of the dictionary $N$.

Each of the vector space methods involves both a filtering step and a checking step. Using these assumptions and considering probabilities of encountering a string with the frequency vector $x$ that satisfies

$$\text{FD}(\text{vect}(q\text{-grams}(p)), x) \leq k(q + [\text{transposition-aware and } q > 1])$$

it is possible to show that the conditional expectation of the computational cost of the checking step is $\beta(p,k)N$, where $\beta(p,k)$ depends on the search pattern $p$ and the maximum allowed edit distance $k$. Then by summing conditional expectations multiplied by probabilities of the pattern occurrence, we obtain that the expected computational cost of the checking step is $\beta(k)N$, for some $\beta(k)$ that depends only on $k$.

From Inequality (46), p. 91, it follows that, in the case of frequency vector tries, the computational cost of the filtering step is

$$O\left((m + k \cdot \mu(q) + 1)^{2k \cdot \mu(q)+1}\right) = o(N),$$

where $\mu(q) = q + [\text{transposition-aware and } q > 1]$. For signature hashing, the cost of the filtering step is $O(2^m)$, where $m$ is the signature size. If we assume that the signature size $m$ is constant and does not change with $N$, then the cost of the filtering step is $o(N)$.

Therefore, the expected computational cost for each of the vector space methods (signature hashing, the unigram frequency vector tries, and the $q$-gram frequency vector tries) is

$$o(N) + \beta(k) \cdot N = O(\beta(k) \cdot N). \tag{43}$$

*C.6.2. Index size.* An index of the method that involves mapping to vector space has three components: a general vector space data structure that stores frequency vectors or signatures, dictionary strings grouped into buckets (all strings in a bucket have the same frequency vector or signature), and pointers to buckets.

Buckets with strings use the same space as the dictionary itself: $O(\lambda N)$, where $N$ is the number of dictionary strings and $\lambda$ is the average dictionary string length. The space used by the vector space data structure and pointers to buckets represents index overhead.

For signature hashing, the data structure is simply an array of size $2^m$, where each element contains a bucket pointer ($m$ is the signature size). Therefore, the index overhead is $2^m P$ bytes ($P$ is the pointer size).

Consider a frequency vector unigram or $q$-gram trie. Let $M(N) \leq N$ be the number of unique $q$-gram (or unigram) frequency vectors. From Theorem C.1, p. 81, it follows that the frequency vector trie uses space $M(N) \cdot (m + 2P + 2)$. Because pointers to buckets use $M(N)P$ bytes, the total index overhead is

$$M(N) \cdot (m + 3P + 2) = O(N)$$

We have verified empirically that $M(N)$ is small in comparison to $N$. For example, a synthesized dictionary with 3.2 million strings used in our experiments, produces less than 300 thousand unigram frequency vectors of the size 10 and less than 300 thousand bigram frequency vectors of the size 9. However, we cannot demonstrate this property analytically.

*C.6.3. Estimate of Search Complexity for Frequency Vector Trie.* Let us consider a frequency distance query with the pattern $z$ and the maximum allowed frequency distance $R$. The maximum allowed distance $R$ is an integer. To obtain the search complexity estimate, we calculate the upper bound for the number of nodes $V(z, R, l)$ visited by the search procedure at level $l$. It can be seen that $V(z, R, l)$ is equal to the number of vectors $x$ generated from $z$ by modifying first $l$ elements so that $\text{FD}(z, x) \leq R$. Note that both $z$ and $x$ contain only non-negative integer elements. The total number of all nodes visited during the search is upper bounded by:

$$\sum_{l=1}^{m} V(z, R, l), \tag{44}$$

where $m$ is the length of frequency vector.

LEMMA C.6.

$$V(z, R, l) \leq \binom{l + R}{R}^2$$

PROOF. $V(z, R, l)$ can be upper bounded by the number of possibly negative integer-valued vectors $y$ of the size $l$, such that $\text{FD}(y, \vec{0}) \leq R$. From the definition of frequency distance it follows that vectors $y$ satisfy:

(1) the sum of positive vector elements is less than or equal to $R$;
(2) the sum of negative vector elements is greater than or equal to $-R$.

The number of ways to choose positive vectors elements in Case (1) is equal to:

$$\sum_{i=0}^{R} \binom{i+l-1}{i} = \binom{l+R}{R}.$$

The number of ways to choose negative vector elements in Case (2) is identical to Case (1). Therefore, the total number of vectors $y$ such that $\mathrm{FD}(y, \vec{0}) \le R$ is upper bounded by $\binom{l+R}{R}^2$. $\square$

From Lemma C.6 it follows that the total number of nodes (including internal ones) visited during the search is upper bounded by:

$$\sum_{l=1}^{m} V(z, R, l) \le \sum_{l=1}^{m} \binom{l+R}{R}^2 \le \sum_{l=1}^{m} \frac{(l+R)^{2R}}{(R!)^2}. \tag{45}$$

The summation term in 45 is a monotonically growing function of $l$. Hence, we can upper bound the sum by the corresponding integral (from 2 to $m+1$) to obtain the final estimate:

$$\int_{l=2}^{m+1} \frac{(l+R)^{2R}}{(R!)^2}\, dl = \left. \frac{(l+R)^{2R+1}}{(R!)^2(2R+1)} \right|_2^{m+1} < \frac{(m+R+1)^{2R+1}}{(R!)^2(2R+1)} \tag{46}$$

## D. EDIT DISTANCE ADDENDUM

### D.1. Proof of theorem 2.8

THEOREM. *From Property 2.4, p. 7, and Property 2.7, p. 8, it follows that*

— *For any two strings $p$ and $s$, there exist a script with the minimum cost, i.e., the edit distance from $p$ to $s$ is properly defined.*
— *The generic edit distance described by Definition 2.6 is a metric [Wagner and Fischer 1974]*

PROOF. To prove that $\mathrm{ED}(p, s)$ is a metric, we need to show that $\mathrm{ED}(p, s)$ exists and is a positive definite, symmetric, and subadditive (i.e., satisfies the triangle inequality) function.

From Properties 2.7, it follows that the cost function is non-negative and that only an identity operation has zero cost. Therefore, without a loss of generality, we can focus on edit scripts that do not have identity operations. Thus, if $p = s$, the only optimal edit script (that does not contain identity operations) is empty and has zero cost. If $p \ne s$, from the completeness of the set of the basic edit operations it follows that there exists one or more edit script that transforms $p$ into $s$. All such edit scripts consist of edit operations with strictly positive costs.

Let $\gamma$ be the cost of an arbitrary script that transforms $p$ into $s$. Consider the set of edit scripts $A$ that transform $p$ into $s$ and whose costs are upper-bounded by $\gamma$. $A$ is non-empty and consists of edit operations with positive costs less than $\gamma$. The set of basic edit operations whose costs are upper bounded by $\gamma$ is finite, which proves that $A$ is also finite. Because $A$ is non-empty and finite, the edit script with the minimum (positive) cost exists and belongs to $A$. Thus, $\mathrm{ED}(p, s) > 0$ for $p \ne s$, i.e., the edit distance is positive definite.

Symmetry of edit distance follows from symmetry of the cost function and from symmetry of the set of basic edit operations $\mathbb{B}$. To prove symmetry of edit distance we consider an optimal script $E$ that transforms $p$ into $s$ and a corresponding reverse script script $E_{\mathbf{r}}$ that transforms $s$ into $p$. It is possible to demonstrate that $\delta(E) = \delta(E_{\mathbf{r}})$.

To prove subadditivity, we consider an optimal script $E_1$ that transforms $p$ into $s$, an optimal script $E_2$ that transforms $s$ into $w$, and a composition of scripts $E_1 E_2$ that transforms $p$ into $w$. From $\delta(E_1 E_2) = \delta(E_1) + \delta(E_2) = \mathrm{ED}(p,s) + \mathrm{ED}(s,w)$ and $\delta(E_1 E_2) \geq \mathrm{ED}(p,w)$ it follows that $\mathrm{ED}(p,s) + \mathrm{ED}(s,w) \geq \mathrm{ED}(p,w)$. $\square$

### D.2. Justification of Mor-Fraenkel Method: Transposition-Unaware Method for $k > 1$

Let $p = p_1 p_2 \ldots p_\eta$ and $s = s_1 s_2 \ldots s_\eta$ be a partition of strings $p$ and $s$ that defines an optimal alignment (see Section 2.2). Additionally assume that $\rho_l$ is a starting position of $p_l$ in $p$ and $\tau_l$ is a starting position of $s_l$ in $s$.

In the case of Levenshtein distance, substrings $p_l$ and $s_l$ are either single characters or the empty strings. In addition, $p_l$ and $s_l$ are not non-empty at the same time. If $\mathrm{ED}(p,s) = k$, there are exactly $k$ indices $l_i$ such that $p_{l_i} \neq s_{l_i}$.

Let $A^p = \{\rho_{l_i} \,|\, p_{l_i} \neq \epsilon \text{ and } p_{l_i} \neq s_{l_i}\}$ be the ordered subset of starting positions of non-empty substrings $p_l$ that do not match respective $s_l$, and $A^s = \{\tau_{l_i} \,|\, s_{l_i} \neq \epsilon \text{ and } s_{l_i} \neq p_{l_i}\}$ be the ordered subset of starting positions of non-empty substrings $s_l$ that do not match respective $p_l$.

OBSERVATION D.2. *If we delete non-empty substrings $p_{l_i} \neq s_{l_i}$ from $p$ and non-empty substrings $s_{l_i} \neq p_{l_i}$ from $s$, we obtain two equal strings. This is equivalent to deleting single characters from $p$ and $s$ whose positions belong to the sets $A^p$ and $A^s$, respectively.*

OBSERVATION D.3. *Multisets $D^p = (A^p_1, A^p_2 - 1, A^p_3 - 2, \ldots)$ and $D^s = (A^s_1, A^s_2 - 1, A^s_3 - 2, \ldots)$ satisfy*

$$|D^s| + |D^p| - |D^s \cap D^p| = k \tag{47}$$

### D.3. Computation of Unrestricted Damerau-Levenshtein Distance

Received 00; revised 00; accepted 00

**Algorithm 2** Computation of the unrestricted Damerau-Levenshtein distance between strings $p$ and $s$

```
var $C$: array [0..$|p|$,0..$|s|$] of Integer
var $CP$: array [1..$|\Sigma|$] of Integer
var $i'$, $j'$, $CS$: Integer

for $i$ := 0 to $|p|$ do $C[i,0]$ := i
for $j$ := 0 to $|s|$ do $C[0,j]$ := j

for $i$ := 1 to $|\Sigma|$ do $CP[i]$ := 0

for $i$ := 1 to $|p|$ do begin
    $CS$ := 0
    for $j$ := 1 to $|s|$ do begin
        if $p[i] = s[j]$ then $d$ := 0 else $d$:= 1
        $C[i,j]$ := $\min(C[i-1,j]+1, C[i,j-1]+1, C[i-1,j-1]+d)$

Comment $CP[c]$ stores the largest index $i' < i$ such that $p_{[i']} = c$.
        $CS$ stores the largest index $j' < j$ such that $s_{[j']} = p_{[i]}$.

        $i'$ := $CP[s[j]]$
        $j'$ := $CS$
        if $i' > 0$ and $j' > 0$ then begin
            $C[i,j]$ := $\min(C[i,j], C[i'-1,j'-1] + (i-i') + (j-j') - 1)$
        end
        if $p[i] = s[j]$ then $CS := j$
    end
    $CP[p[i]] := i$
end
output $C[|p|,|s|]$
```