Prefix Tree Indexing for Similarity Search and Similarity Joins on Genomic Data

Astrid Rheinländer, Martin Knobloch, Nicky Hochmuth and Ulf Leser

Humboldt-Universität zu Berlin
Department of Computer Science
Berlin, Germany

Abstract. Similarity search and similarity join on strings are important for applications such as duplicate detection, error detection, data cleansing, or comparison of biological sequences. Especially DNA sequencing produces large collections of erroneous strings which need to be searched, compared, and merged. However, current RDBMS offer similarity operations only in a very limited and inefficient form that does not scale to the amount of data produced in Life Science projects.
We present PETER, a prefix tree based indexing algorithm supporting approximate search and approimate joins. Our tool supports Hamming and edit distance as similarity measure and is available as C++ library, as Unix command line tool, and as cartridge for a commercial database. It combines an efficient implementation of compressed prefix trees with advanced pre-filtering techniques that exclude many candidate strings early. The achieved speed-ups are dramatic, especially for DNA with its small alphabet. We evaluate our tool on several collections of long strings containing up to 5,000,000 entries of length up to 3,500. We compare its performance to *agrep*, *nrgrep*, and user-defined functions inside a relational database. Our experiments reveal that PETER is faster by orders of magnitudes compared to the command-line tools. Compared to RDBMS, it computes similarity joins in minutes for which UDFs did not finish within a day and outperforms the built-in join methods even in the exact case.

1 Introduction

Similarity search and similarity join on string data has become a topic of interest in the past years [4, 10]. Applications arise in duplicate detection [15], error correction [13] and data cleansing [5], to name only a few. They are also of uttermost importance in the Life Sciences. The characteristics of all organisms are coded in their genomes, which can be represented as a very large string over an alphabet of four letters. Approximately searching DNA sequences is important in virtually all fields of modern genomics. In this paper, we will use ESTs as our running example. *ESTs(Expressed Sequence Tags)* are short DNA sequences with lengths mostly in the region of 300 to 800 bases that are commonly used to identify genes and their localization on a chromosome. However, to be cost-effective, ESTs are obtained by a single sequencing pass which yields in an estimated error rate of 1% [9]. Thus, few differences in ESTs often are simply due to errors created by the sequencing process, which implies that searching and joining EST data sets should always be carried out approximately rather than exactly. Since the EST sets that are considered go in the millions[1], efficient execution of such similarity operations is crucial.

A large number of tools has been developed by the bioinformatics community to speed-up similarity search [2, 20]. Almost all of them focus on computing local alignments [6] and use heuristics to achieve performance – at the cost of accuracy. In contrast, we aimed at developing an algorithm that supports global alignment (i.e., comparison of entire strings and not of substrings) and that is exact. Furthermore, we want our algorithms to be operations inside a relational database. The reason for this decision is that advanced analysis of sequences often depends on the availability of additional information (such as gene function, genomic annotation, biological pathways etc.), which nowadays is mostly maintained in RDBMS [11]. Furthermore, our intention is to provide a universal data structure for similarity operations not restricted to the Life Sciences.

In this paper, we present PETER, an indexing algorithm for scalable approximate search and approximate join operations based on Hamming distance or edit distance. PETER builds on a compressed prefix tree. One advantage of prefix tree indexing is that the complexity of search queries only depends on the depth of the tree, i.e., the maximal length of the indexed strings, and not on the number of indexed strings. Joins between sets of trees can be implemented efficiently by computing the intersection of two prefix trees. However, it is not trivial to sustain these advantages when moving from exact to similarity operations [16]. To this end, we refine algorithms for similarity search in prefix trees with various pruning and filtering techniques. Since we focus on retrieving very similar strings, these reduce the search space significantly; this focus also allows us to use a special alignment method (k-banded alignment) which is much faster than normal edit distance computation. We show that our tool outperforms the Unix command line tools `agrep` and `nrgrep` by magnitudes and also show that it enables efficient similarity based search and join queries on large string collections inside a RDBMS. At the downside, a restriction of our tool is that it is only efficient for searching highly similar strings; however, this is the predominant requirement in most applications we are aware of.

Compressed prefix trees [16], k-banded alignment [3] and the filtering techniques we apply [1, 4, 16] have been published before. However, this is the first work that combines these different ideas to speed up similarity search into a single, homogeneous algorithm supporting both similarity search and similarity joins. It is also the first work we are aware of that persistently integrates such a method into an RDBMS, thus offering its capabilities to SQL users.

[1] The largest collection of publicly available EST sequences is dbEST [14] with more than 60 million EST sequences from 1745 organisms as of May 2009.

Our paper is structured as follows. Chapter 2 contains an introduction to our data structures and to similarity search in general. In Chapt. 3 we describe our techniques for efficient similarity operations on compressed prefix trees. Chapter 4 gives details on the implementation. We present experimental results in Chapt. 5. Related work is discussed in Chapt. 6 and Chapt. 7 concludes the paper.

2 Preliminaries

Let Σ be an alphabet (we will usually use $\{a, c, g, t\}$). We use s, with subscripts if required, to denote strings in Σ^*. Let $n = |s|$ be the length of s. A substring of s, denoted by $s[i \dots j]$, starts at position i and ends at position j. We call $s[1 \dots j]$ prefix, $s[k \dots |s|]$ suffix and $s[i..j], (1 \leq i \leq j \leq |s|)$, infix of s. Any infix of length q is called q-gram. For a fixed q, s contains $m = n - q + 1$ q-grams. A pair $(i, s[i, \dots, i + q - 1])$ is called *positional* q-gram [18].

2.1 Similarity Measures

Similarity-based operations must be based on a concrete similarity measure. PETER supports Hamming distance and edit distance.

Definition 1 (Hamming distance). *The Hamming distance $d_{hd}(s_1, s_2)$ of two strings s_1, s_2 of equal length is the number of mismatching characters in s_1 and s_2: $d_{hd}(s_1, s_2) = |\{i | s_1[i] \neq s_2[i]\}|$. We say two strings are within Hamming distance k if $d_{hd}(s_1, s_2) \leq k$.*

Obviously, computing the Hamming distance of two strings with $|s_1| = |s_2| = n$ is possible in $O(n)$. However, Hamming distance is only defined for strings of equal length, and also an inappropriate measure in most bioinformatics applications. There, we are mostly interested in the minimal number of operations that turn one string into the other, called the edit distance (or Levenshtein distance).

Definition 2 (Edit distance). *The edit distance $d_{ed}(s_1, s_2)$ of two strings s_1, s_2 with $|s_1| = n, |s_2| = m$ is the minimal number of insertions, deletions, or replacements of single characters needed to transfrom s_1 into s_2. We say two strings are within edit distance k, if $d_{ed} \leq k$.*

Using dynamic programming, the edit distance can be computed $O(|s_1| * |s_2|)$ [13, 19]. However, faster computation is possible when one is only interested in highly similar strings. The *k-banded alignment* algorithm finds the edit distance of two strings with edit distance of at most $2k$ in $O(k * max\{|s_1|, |s_2|\})$.

Definition 3 (k-banded alignment). *The k-band of an edit distance matrix M is defined as: $M[i, j] \in k\text{-}band \Leftrightarrow |i - j| \leq k$. If s_1 and s_2 are within edit distance k, their optimal alignment path must lie in the k-band of M. Thus, all cells of M that are not in the k-band can be ignored [3].*

2.2 Similarity Operators

There are various forms of defining similarity operator [13]. We support two such operations: similarity search and similarity join.

Definition 4 (Similarity search). *Given a string s, a set S of strings, and a threshold k, $ssearch(s, S)$ returns all $s' \in S$ for which $d(s, s') \leq k$.*

Definition 5 (Similarity join). *Given two sets S_1, S_2 of strings and a threshold k, $sjoin(S_1, S_2)$ returns all pairs $(s_1, s_2), s_1 \in S_1, s_2 \in S_2$ for which $d(s_1, s_2) \leq k$.*

As described before, we support Hamming and edit distance as distance function. Note that both operations naturally also support exact search and exact joins, simply by setting $k = 0$ for either distance measure. We will see in Chapt. 5 that even exact joins on large collections of long strings are considerably faster with PETER than using hash or merge joins.

2.3 Compressed Prefix Trees

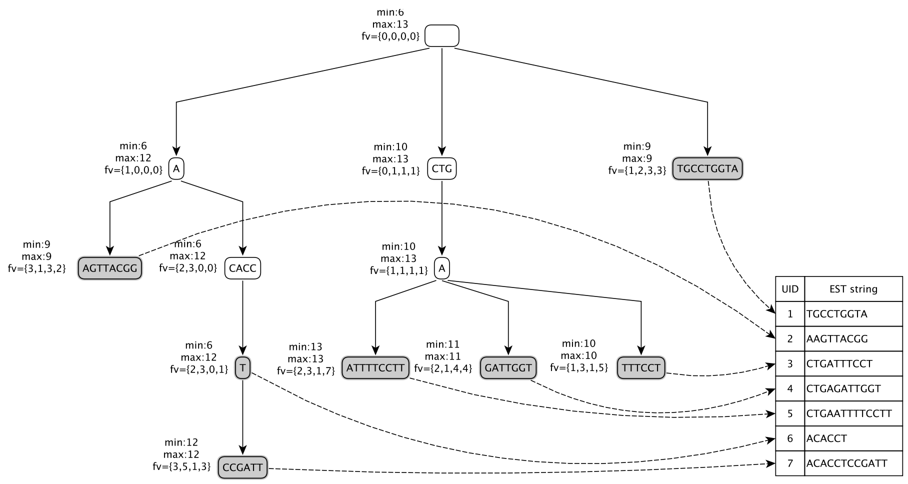
Our fundamental data structure are compressed prefix trees [12], built on top of a set of strings. Let R be a set of tuples (s, ID) where $s \in \Sigma^*$ and ID is a unique identifier for s.

Definition 6 (Prefix tree index). *A compressed prefix tree index T for R is a rooted, directed tree that meets the following conditions:*

1. *Every node x is labeled with a sequence of characters $c_i \in \Sigma$ of length $l \geq 1$. The labels of any two children y, z of the same node x start with a different character.*
2. *Every string $s \in R$ maps to some node $x \in T$ such that the concatenation of all labels from T's root to x exactly is s. We call x string node and assign the corresponding ID to x. If a particular string occurs several times in R, all corresponding IDs are assigned to x.*
3. *(Compression of suffixes). Let x be the root of a subtree formed by a linear chain of children $x_1, \ldots, x_m$, where solely x_m is a string node and has no further children. Then, $x, x_1, \ldots, x_m$ are merged to a single node x' whose label is the concatenation of their labels. The ID of x_m is assigned to x'.*
4. *(Compression of infixes). Let x be the root of a subtree formed by a linear chain of children $x_1, \ldots, x_m$, where no node is a string node and only x_m has more than one child. Then, $x, x_1, \ldots, x_{m-1}$ are merged to a single node x' whose label is the concatenation of their labels.* □

Conceptually, nodes may have labels of arbitrary length. Technically, we store labels of string nodes without children (e.g., unique suffixes in R) in two parts: A small part (usually 16 characters) is stored inside the node. The rest of the suffix is stored in an extra file (see Sect. 4.2 for details). Furthermore, we attach further information to every node, namely minimum/maximum string lengths and a frequency vector (see Sects. 3.2 and 3.3).

Figure 1 shows an example of a compressed prefix tree. It has simple nodes (e.g., “A”), a compressed infix node (“CTG”), and a compressed suffix node (“TGCCTGGTA”).



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Fig. 1. Compressed prefix tree. Grey nodes are string nodes. Min/max specify minimum and maximum string lengths, fv denotes the frequency vector (see text).

3 Similarity Operations on Prefix Trees

To our knowledge, similarity search in prefix trees based on pre-order traversal was first studied in [16]. This method computes the similarity for a given pattern p to all strings indexed in a prefix tree T while traversing T. While this does not change the worst-case complexity of searching, large savings are achieved when the indexed strings share many prefixes as these prefixes need to be compared only once to a prefix of the pattern. Thus, the method is very well suited for small alphabets and for very large string collections, as these properties increase the number and average lengths of shared prefixes.

Shang et al. described shared prefix analysis in [16]. In the following, we very briefly repeat the general idea but concentrate on our various extensions which greatly increase effectiveness: switch from global alignment to k-banded alignment, addition of filtering techniques for further search space pruning, and extension of the method to also allow similarity joins. Filtering is performed at two stages. For edit distance, we use a combination of length and frequency filtering to prune whole subtrees. Whenever we reach a leaf, we apply a q-gram pre-selection to suffixes. For Hamming distance, we only use length and frequency filtering.

All extensions are described below. For space reasons, we focus on similarity search and only briefly mention changes necessary to compute similarity joins. Consider a compressed prefix tree T for a set of strings R. Let p be a search pattern and S another indexed set of strings. Let t be a node in T. By slight abuse of notation, we use t both for the node and for the concatenation of labels from root to the node t. Let k be a user-defined threshold for similarity operations.

3.1 Shared Prefix Pruning

Clearly, t represents the shared prefix $s[1 \ldots |t|]$ of a set of strings all of which must be descendants of t. Note that both Hamming and edit distance to p can only grow with growing prefix length. Thus, the following holds:

1. Hamming-search: If $d_{hd}(t, p[1 \ldots |t|]) > k$, then all strings below t can be pruned. Thus, traversal does not descend further from t.
2. Hamming-join: Both trees, T and S, are traversed simultaneously. Only nodes with $|s| = |t|$ need to be compared to each other. Let $s \in S$ have the same label length as t. If $d_{hd}(t, s) > k$, then the subtrees starting at t in T and s in S can be pruned.
3. Edit-search: Let p be aligned horizontal and t vertical in the k-banded distance matrix. All strings in the subtree below t share the same prefix t and thus also share rows 0 to $|t|$ in the matrix. If row $|t|$ contains only values larger than k, then no string below t can have a smaller edit distance to p. This subtree can be ignored for further search.
4. Edit-join: Again, all strings below s share the same prefix. The same arguments as for search hold, except that now we compare to a shared prefix in S instead of a single p. Additionaly, the subtree also can be pruned if any row contains only values larger than k.

3.2 Length Filtering

Trivially, t is a candidate for p regarding Hamming distance only if it is of equal length as p. With respect to edit distance, t and p are worth examining only if $|t| - |p| \leq k$. To quickly check this property, we store two additional attributes at every node - the minimum (min) and maximum (max) string length of all strings below that node. If $(|max(t)+k| < |p|) \vee (|min(t)-k| > |p|)$ holds, then no string below t can be edit-similar to p. Similarly, if $(|max(t)| < |p|) \vee (|min(t)| > |p|)$ holds, no string below t can be Hamming-similar to p. Thus, traversal will not descend further. The same argument applies conceptionally to similarity join, when p is replaced with a shared prefix in the joined tree S.

3.3 Frequency Distance Filtering

Aghili et al. [1] proposed frequency distance based filtering to reduce candidate sets in similarity searching on strings. Consider a string $s \in \Sigma^*$. The corresponding frequency vector $fv(s)$ of s consists of $|\Sigma|$ components, where component i counts the number of occurrences of $x_i \in \Sigma$ in s.

Definition 7 (Frequency distance). *For $\{s_1, s_2\} \in \Sigma^*$, the frequency distance $fD(s_1, s_1)$ is the minimum of the necessary number of applications of $+1$ or of -1 needed to transform $fv(s_1)$ into $fv(s_2)$.*

Example 1. Consider $s_1 =$ acacctccgatt, $s_2 =$ acacatccgaaa with Hamming distance $d_{hd} = 3$ and edit distance $d_{ed} = 3$. The corresponding frequency vectors for s_1, s_2 are $fv(s_1) = [3, 5, 1, 3]$ and $fv(s_2) = [6, 4, 1, 1]$. To transform $fv(s_1)$ into $fv(s_2)$, we need to add $3*(+1)$ for character a, $1*(-1)$ for c, and $2*(-1)$ for t. The frequency distance sums up to $fD(s_1, s_2) = min(|3*(+1)|, |3*(-1)|) = 3$.

Actually, frequency distance is a lower bound to Hamming and to edit distance [1]. Thus, evaluation of frequency vectors gives a third method to stop traversal of the index tree. We prune a subtree starting at t, if $fD(t, p) - (|t| - |p|) > k$ on both similarity search settings. Extension to join is straight-forward.

3.4 Q-Gram Filtering

Indexing methods based on q-grams are well-known to restrict search spaces efficiently for edit distance operations. They take advantage of the observation that two strings are within a small edit distance if they share a large number of q-grams [17]. Actually, the number of matching q-grams acts as another bound to edit distance [4].

Definition 8 (Mismatching q-grams). *Let Q_{s_1}, Q_{s_2} be sets of positional q-grams of s_1 and s_2, respectively. s_1 and s_2 are within edit distance k, iff $|Q_{s_1} \cap Q_{s_2}| \geq max(|s_1|, |s_2|) - 1 - (k-1)*q$. A string s_1 is not within edit distance k to s_2, if Q_{s_1} contains at least $|Q_{mis}| = (|s_1| - q + 1) - (max(|s_1|, |s_2|) - 1 - (k-1)*q)$ positional q-grams that are not contained in s_2.*

The choice of q commonly depends on the average string length l. In this work, we follow [13] and use $q = log_{|\Sigma|}(l)$. Unlike [4], we do not use q-grams for indexing, but for suffix pre-selection. As shown in Fig. 2, nearly 90% of all indexed strings in our evaluation data set have large, unique suffixes. Determining the k-banded edit distance for the whole string needs $m * (2k + 1)$ computations for the suffix (with m smaller, but often not much smaller than $|s|$), whereas the computation of positional q-grams for that suffix takes only $m - q + 1$ operations. Thus, the costs for comparing mismatching q-grams are on average smaller than computing the edit distance immediately. Therefore, when a leaf with suffix $s[x \ldots |s|]$ of length m is reached, we compute all positional q-grams for this suffix. For similarity searches, we also extract the suffix of length m from the search pattern and evaluate Q_{mis} on both sets. For approximate joins, we evaluate Q_{mis} only if we have reached a suffix leaf in both relations. We do not apply q-gram filtering to Hamming distance queries as the costs for building and evaluating q-grams are higher than comparing the suffixes directly.

4 Implementation

In this section, we describe implementation details on the primal functions of PETER. All algorithms can be executed standalone from the command line, as operators inside a database, or as a library in other C++ programs. Our implementation allows for both indexing database relations and flat files. We provide functionality for index building, searching, and insertion, deletion or modification of indexed strings. PETER also contains an index optimization routine that physically rearranges the trees and the suffix files in order to decrease the number of disk page accessions during tree traversal. This method should be called after extensive changes to the index and the string set to prevent index degradation. Next to approximate search and join operations, we support exact searches and joins as well as range queries, the SQL 'LIKE'-operator and prefix-based searches.

4.1 Algorithms

Search. Algorithm 1 shows the pseudocode for similarity searching in a prefix tree index. PETER essentially performs a depth-first traversal of the prefix tree applying all pruning techniques presented in Chapt. 3. Before descending from a node, we apply length and frequency filters. We also prune if we exceed the allowed distance threshold. In case of edit distance searches, we check for every reached string node whether the number of mismatching positional q-grams exceeds Q_{mis}. The function *getDistance* checks a flag whether a Hamming or edit distance search is performed and computes the new distance. When a match was found, the pair of matching EST objects (each pair consists of the ESTs and their UIDs) is added to the result set. Finally, the result set is returned to the user and by default printed to `stdout`.

Join. When using the join operator (see Algorithm 2) on two relations, conceptionally the intersection of two trees is computed. Both index trees T and S are traversed concurrently, such that the tree with less nodes is traversed first. Tree sizes are checked at startup. Length and frequency filtering are applied as in the search algorithm. Global variables are used to store the number of processed characters for the currently expanded strings in both trees. If we reach a string node in tree T (or S, respectively), we fetch the complete string represented by this node and perform a call to the search function, with the string as pattern, the remaining subtree S' of S (T' of T) and the current distance value as parameters. We continue the distance computation for the next untreated characters in the string and S' (T'). The result set contains all pairs of matching EST objects and is constructed through the search algorithm. Finally, it is returned to the user and printed to `stdout`.

Algorithm 1 searchTree(Node x, EST p, int k, int d)

```
 1: if $isLeaf(x) \wedge getEditDistance() \wedge !passesQGramFilter(x)$ then
 2:     return
 3: end if
 4: $newDistance \leftarrow getDistance(x, p, d)$
 5: if $newDistance > k$ then
 6:     return
 7: end if
 8: if $hasID(x)$ then
 9:     $addMatchingESTsToResultSet(s[1 \ldots x], p)$
10: end if
11: for all children y of x do
12:     if $passesLengthFilter(y, p, k) \wedge passFrequencyFilter(y, p, k)$ then
13:         $searchTree(y, p, k, newDistance)$
14:     end if
15: end for
```

Algorithm 2 joinTree(Node x, Node y, int k, int d)

```
 1: $newDistance \leftarrow getDistance(x, y, d)$
 2: if $hasID(x)$ then
 3:     $pattern \leftarrow getString(1, x)$
 4:     $searchTree(y, pattern, k, newDistance)$
 5: else if $hasID(y)$ then
 6:     $pattern \leftarrow getString(1, y)$
 7:     $searchTree(x, pattern, k, newDistance)$
 8: end if
 9: for all children x' of x do
10:     for all children y' of y do
11:         if $passesLengthFilter(x', y', k) \wedge passesFrequencyFilter(x', y', k)$ then
12:             $joinTree(x', y', k, newDistance)$
13:         end if
14:     end for
15: end for
```

4.2 Index Structure

Our index structure is physically stored in two files, the prefix tree and a suffix file. In the prefix tree file, all nodes are stored contiguously in pre-order arrangement. Nodes may have variable size. Each one consists of a node id, information on the node type (string node or not), its label, references to all children, max. and min. lengths, and a frequency vector. If a node is a string node, it also contains the respective ID; if the node maps to more than one ID, we store a reference to an index-sequential file that contains all IDs instead.

String nodes also store the length of the remaining suffix and its prefix (of the suffix). As explained previously, this suffix can be quite large; actually, those suffixes form the bulk of the size of the entire data structure. We therefore decided to store all but short prefixes of the suffixes in an extra file, which allows us to keep the prefix tree itself in main memory even for very large string collections. Actually, the tree file for TX (5,000,000 strings, see Fig. 2) has a size of 549 MB on disk and 943 MB when the prefix tree is kept in main memory. Suffixes of the suffixes are stored in an external file referenced from string nodes. Suffix accession follows the *lazy evaluation* paradigm both for q-gram evaluation and character comparison: if a suffix node is reached, the internal suffix is examined first. External suffixes are only loaded when needed. Very often, this method allows to take decisions without accessing the suffix file.

Insert operations to the index are executed sequentially. First, our insert algorithm searches by depth-first traversing the index tree the appropriate insert position for the new EST string. We update the values for min/max for all ancestors while descending. Existing nodes might be modified, all newly created nodes are appended to the end of the index file. In case the index file gets too fragmented, we manually launch an optimizing routine that rearranges the index and suffix file in preorder accession.

4.3 Integration into RDBMS

We integrated our programs as a shared library into a commercial RDBMS (name omitted) using its built-in extension capabilities. These include user-defined indexes (used for prefix tree) and user-defined table functions necessary to implement similarity joins. Integrating user-defined functions and indexes consists of two parts: The first part is the program code, compiled as a shared library and saved at a specific position in the file system. The second part involves declarations and definitions directly executed on the server, including a reference to the library.

When index and query functions are accessed for the first time in a session, the server determines the location of the shared library. A listener process invokes a session-specific agent and passes over the call including procedure and library name and any parameters, if present. The agent loads the library and runs the desired function, that, in our case, in turn opens the index (and later on the suffix file if required). Any return values are passed back via the agent. Throughout the session, this agent remains alive, which implies that initialization costs for the agent emerge only once.

However, only the code is kept in memory, while any data loaded during executing of the call are discarded by the agent. Caching or buffering of user-loaded data is not supported. This is a severe drawback of the extension mechanism, since it implies that, in our case, the entire prefix tree is loaded again for every single call of a similarity search. As we will see in the next Chapter, this method incurs a large penalty on any user-defined index compared to the server's build-in methods.

5 Experiments

We use ESTs (see Introduction) to evaluate the performance of PETER both for similarity and for exact operations ($k = 0$). To this end, we extracted a subset from dbEST as of 28.05.2009. We fixed $|\sigma| = 4$ and removed all sequences containing characters other than A,C,G or T. Figure 2 shows properties of the sets we used during the evaluation. T_i consists of EST sequences from the i-th dbEST archive, T_{ij} consists of a subset j of randomly chosen EST strings from T_i, TX consists of randomly chosen EST strings from the dbEST archives 20 to 26. We show results for varying numbers of indexed strings,for varying k, and for each filter technique (see Chapt. 3) in isolation. Index creation and optimization was performed in advance and is not included in the measured times (but see Fig.2). We observed that the time for index creation grows, as expected, linear with the number of indexed strings.

We compare the performance of PETER against two competitors: The Unix tools command line tools `grep`, `agrep`, and `nrgrep`, and build-in or user-defined functions inside the RDBMS. We also tried to compare to other recently published methods, such as [5, 21], but none of these is available for download. In particular, we acknowledge that comparison against Unix command line tools are not completely satisfactory, as our method first builds an index of the set to be searched. Therefore, it is more suited for searching the same set of strings multiple times. However, we will show that index creation time is leveraged already for very few searches (see Section 4.3). Note that [16] also compared against `agrep`.

All experiments were performed on a Pentium-M 740 processor with 2 GB RAM. For each experiment, we report the average of ten runs.

Set	# EST strings	avg. string length	min/max length	# tree nodes	# ext. suffixes
T_1	307,542	348	14/3,615	589,062	293,764
T_2	736,305	387	12/3,707	1,482,709	689,590
T_{2a}	368,152	382	12/2,774	711,632	352,872
T_{2b}	184,076	385	22/2,774	349,329	177,846
T_{2c}	92,038	383	25/2,774	171,964	89,198
T_{2d}	46,019	381	28/2,774	84,954	44,716
T_{2e}	23,009	373	31/ 878	42,375	22,366
T_3	10,000	536	16/3,707	16,310	8,774
TX	5,000,000	359	14/3,247	10,478,214	4,834,231

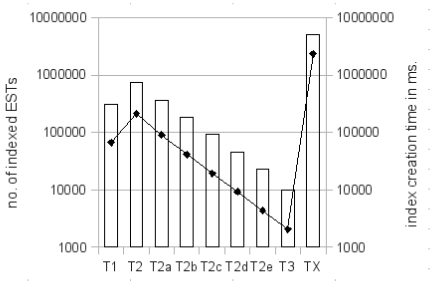


Fig. 2. Left: Properties of EST sets. Right: Index creation (line) wrt. set size (bar).

5.1 Effect of Pruning Strategies

We evaluated the effects of length, frequency, and q-gram filtering individually and in all possible combinations using sets T_1 and T_3. For search queries, we performed searches for all EST strings taken from T_3 in T_1 with different similarity thresholds. For joins, we computed $T_1 \bowtie T_3$.

Search results are shown in Fig.3. Overall, frequency filtering did not lead to significant runtime improvements. We suppose that this is caused by the small alphabet which makes the compared vectors very small. Length filtering leads to improvements for Hamming distance searches in the range of 5% for $k = 1$ growing up to 76% for $k = 8$. Looking at edit distance searches, length filtering performed even better, with improvements in the range of 10% for $k = 1$ growing up to 86% for $k = 8$. Interestingly, q-gram filtering in edit distance searches improved the execution time significantly only for $k = 1$ (10%) and $k = 3$ (13%). But combining length and q-gram filtering for small k improves average execution times in the range of 18% for k=1, for $k = 2$ up to 58% and 81% for $k = 3$. There is also a clear tendency that runtime improvements achieved by filtering increase with increasing similarity thershold, i.e., they are the more effective, the more differences are allowed.

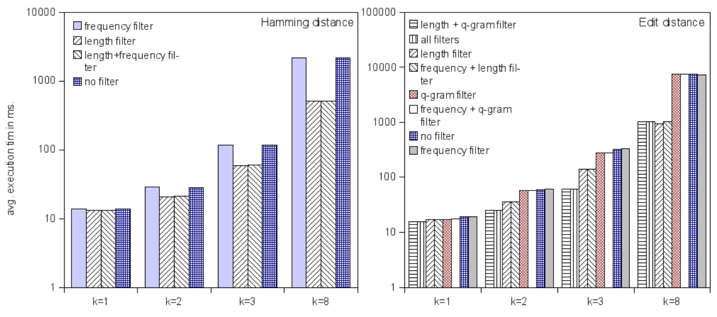


Fig. 3. Search execution time of $p \in T_3$ with $k \in \{1, 2, 3, 8\}$ in T_1 wrt. filters (log-scale).

In terms of join algorithms, frequency filtering again only has a neglible impact(see Fig.4). For joins on Hamming distance, we observed that length filtering improved the join execution time from 18% ($k = 0$) up to 40 % ($k = 3$). For edit distance joins, single q-gram filtering leads to improvements in the range of 50% for $k = 0$. For growing k, standalone length filtering was more effective. Again, speed-ups roughly correlate with allowed differences.

Overall, a combination of length and q-gram filtering seems to be the best configuration. Therefore, in all following experiments with PETER we always used length filtering for Hamming distance and a combination of length and q-gram filtering for edit distance.

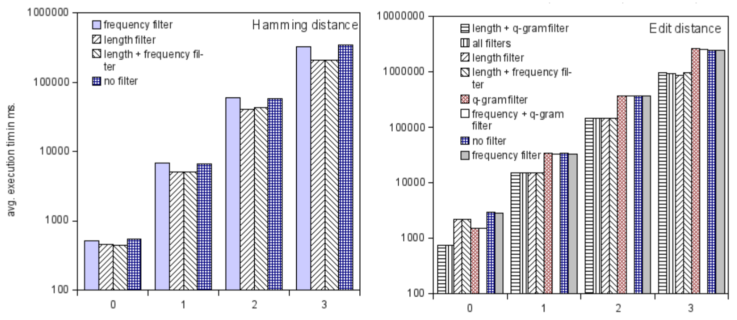


Fig. 4. Join execution time of $T_1 \bowtie T_3$ with $k \in \{0, 1, 2, 3, 8\}$ wrt. filters (log-scale).

5.2 Performance of Search

We compared the execution times of PETER for Hamming and edit distance for various tresholds to Unix command line tools. We used `grep` for $k = 0$, and `agrep` and `nrgrep` for $k \in \{1, 2, 3, 8\}$, respectively. First, we performed individual searches for each pattern $p \in T_3$ in the indexed EST set T_1. When searching with the Unix tools all searches were started to match only complete strings to the given pattern. As `agrep` is bounded with a maximum pattern length of 32 characters, we used `nrgrep` to handle longer patterns.

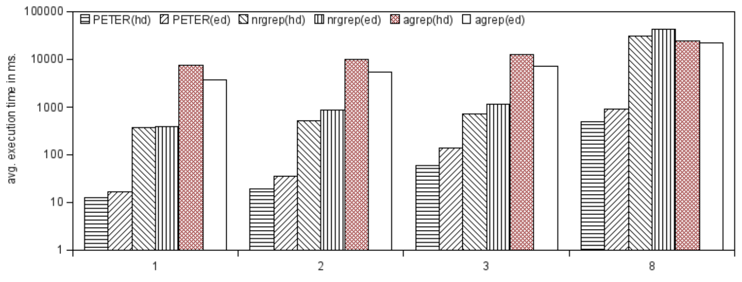
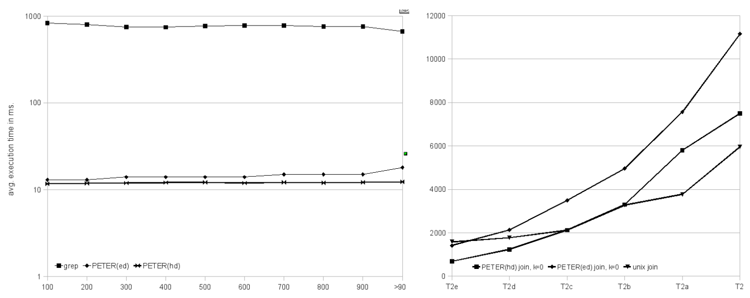


Fig. 5. Search in PETER vs. Unix tools for $p \in T_3, \quad k \in \{1, 2, 3, 8\}$ (log-scale).

For exact search, we outperform `grep` significantly with a factor of 63 for Hamming distance and a factor of 50 for edit distance scoring enabled. As shown in Fig.6(a), the impact of the pattern length was negligible for exact searches in all tested methods. Figure 5 contrasts the average execution times of inexact searches to `agrep` and `nrgrep`. For very short patterns, we outperform `agrep` with factors in the range of 640 for $k = 1$ up to 1063 for $k = 8$ on Hamming distance and a factor up to 450 for edit distance constraints. When searching with patterns of arbitrary length, we are up to 60 times faster than `nrgrep` for Hamming distance and up to 45 times faster for edit distance.

We observed a small influence of pattern lengths with growing tresholds (data not shown). Searching for patterns of lengths 200 to 600 took slightly longer than searching for shorter or longer patterns. This is not surprising as strings with lengths in this range make up most of all strings in T_1. Searching with Hamming distance constraints has always better response times than edit distance, in the range of 5% ($k = 0$) to 65 % with growing k. This is caused by costs for initializing and computing the edit distance matrix.

Even if we add the costs for index creation to the evaluation, PETER amortizes quite fast. For example, if we run multiple Hamming distance (edit distance, respectively) searches in T_1 with $k = 1$, it takes only 10 (15) searches to outperform the cumulated runtimes of `agrep`. Compared to `nrgrep`, it takes 125 (105) Hamming distance (edit distance) queries until PETER is profitable.



(a) Search $p \in T_3$ in T_1 (b) Join $T_1 \bowtie T_{2i}$, $i \in \{a, b, c, d, e\}$

Fig. 6. Avg. execution times for exact searches (log-scale) and joins, k=0.

5.3 Performance of Similarity Join

We compared the execution time of a natural join on $T_1 \bowtie T_{2i}$ in PETER to the Unix command `join`. All joins were highly selective even for large tresholds as indicated in Fig.7. Since `join` expects sorted flat-files as input, we performed a preprocessing step on the corresponding EST sets that is not included in the execution time of `join`. We thought this to be fair, as index creation times also are not included in the measurements with PETER. `Join` almost always outperforms our algorithm with $d_{ed} = 0$. For Hamming distance $d_{hd} = 0$ PETER beats `join` if the joined sets differ in size with a factor of at least 1.7, as presented in Fig.6(b). Both observations are not surprising since an exact join on sorted input only requires to linearly scan both files. For approximate joins, we are not aware of any Unix command line tool that could handle this problem. Comparing edit distance to Hamming distance joins, the latter always performed in a range of 30% to 60 % better, mostly dependent on the given treshold (see Fig.8). We observe an exponential growth of join execution times with respect

Join	$\lvert A \times B\rvert$	$\lvert A \bowtie B\rvert$ $k=0$	$\lvert A \bowtie_{hd} B\rvert$ $k=1$	$\lvert A \bowtie_{hd} B\rvert$ $k=3$	$\lvert A \bowtie_{ed} B\rvert$ $k=1$	$\lvert A \bowtie_{ed} B\rvert$ $k=3$
$T_1 \bowtie T_2$	226,444,712,310	299	514	841	941	2,552
$T_1 \bowtie T_{2a}$	113,222,202,384	187	326	374	239	463
$T_1 \bowtie T_{2b}$	56,611,101,192	128	214	236	152	301
$T_1 \bowtie T_{2c}$	28,305,550,596	78	127	178	75	232
$T_1 \bowtie T_{2d}$	14,152,775,298	42	71	91	46	165
$T_1 \bowtie T_{2e}$	7,076,233,878	28	41	55	33	106
$T_1 \bowtie T_3$	3,075,420,000	1,048	1,053	1,059	1,058	1,082
$T_{2e} \bowtie T_3$	230,090,000	54	70	106	94	258
$T_1 \bowtie TX$	1,537,710,000,000	37	115	372	354	992

Fig. 7. Join cardinalities for Hamming distance ($\bowtie_{hd}$) and edit distance ($\bowtie_{ed}$) join.

to the treshold although the result sets don't grow exponentially. The reason for this is that the search space increases exponentially with growing k. While tree traversal, PETER descends further as k grows and for every additional node, that is reached in T, there are $|\sigma|$ additional subtrees examined in S.

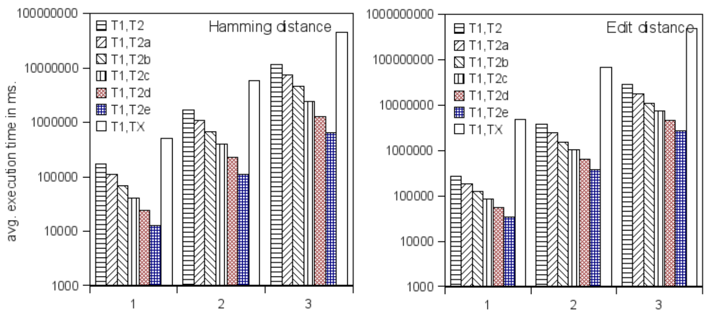
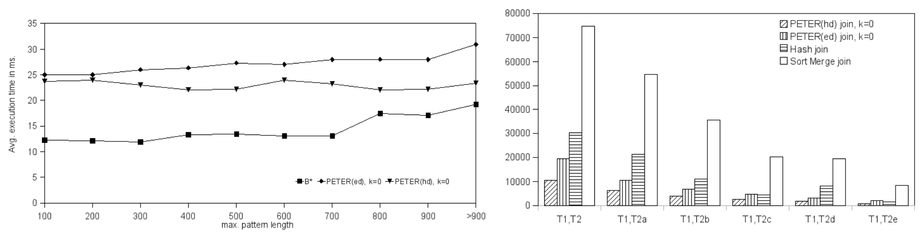


Fig. 8. Similarity join in PETER for $k \in \{1, 2, 3\}$ on $T_1 \bowtie T_{2i}$, $T_1 \bowtie TX$ (log-scale).

5.4 Performance of PETER Inside a RDBMS

We compared PETER's performance against exact and similarity-based search and joins inside the RDBMS. For searching, we performed single `SELECT` queries on the B*-indexed relation T_1 for each EST string in T_3. At all times and for different pattern length, the built-in `SELECT`-operator achieves better runtimes than a prefix tree based search, see Fig.9(a). Factors vary dependent on the pattern length, in a range of 2 ($|p| \leq 400$) to 1.3 ($|p| \geq 800$). There are mostly two reasons for this result. First, the operations in the prefix tree index are handled via the extension interface which produces overhead for every call. Second, the extension interface does not allow caching of data. While the internal implementation uses the internal buffer pool of the database to cache the most important parts of the B*-index, this is not possible for user-defined indexing. Given these severe drawbacks, it is noteably that PETER is only so little slower. Although the page cache of the OS significantly speeds up the response times of subsequent queries on the command-line (25–30% speedup on avg.), we could not observe this effect inside the RDBMS (5–8% speedup on avg.). This indicates that a more tight integration into the database kernel could give a data structure like PETER significant advantages over B*-indexes (for large sets of long strings).



(a) Search (b) Join

Fig. 9. PETER vs. RDBMS built-in operators for $k = 0$. For result sizes see Fig.7.

Regarding joins, we computed $T_1 \bowtie T_{2i}$ as a Hash join and as a Sort-Merge join and compared these results to PETER. Joins on the prefix tree index always outperformed both Hash join (with factors between 1.5 and 4) and Sort-Merge join (with factors 3.8 to 10), see Fig. 9(b). Note that the problem of caching is not a severe one here, as computing the join requires to load both indexes only once. We find PETER's performance for exact strings quite remarkable as they – for large sets of long strings – beat the highly-tuned joins of a commercial database system.

As there are no built-in functions for similarity operations inside the database, we implemented them as user-defined functions (UDF) in the database's programming language. The edit distance function computes the k-banded alignment score for two strings. Length filtering is included in both UDFs. We compared the execution time of UDF-based similarity search and joins to PETER. Fig.10 shows that PETER for similarity searches performs better by an order of magnitude than using just UDFs. For Hamming distance search, prefix tree indexing leads to a runtime improvement factor of about 520, for edit distance searches of about 890. We also tried to perform similarity joins for $k = 1$ with UDFs on $T2_e \bowtie T3$, but as the join operations did not finish within a day, we aborted the execution. Similarity joins with prefix trees finished for $k \in \{1, 2\}$ in less than one minute.

6 Related Work

Prefix trees, compressed or uncompressed, have applications in many areas. They are well known for representing multiple search patterns in exact pattern matching [6], but have also been shown to perform well in frequent itemset mining algorithms [7] or set joins [8].

Shang et al. [16] were the first that extended prefix trees with dynamic programming techniques to perform inexact string matching. They have shown that

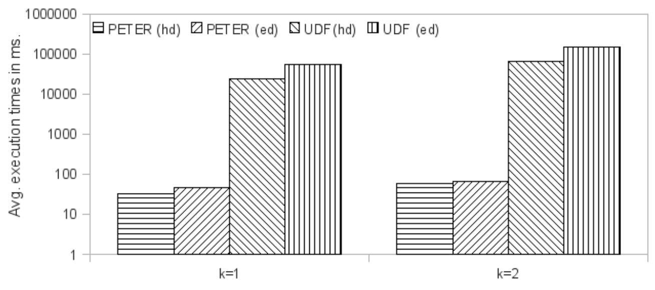


Fig. 10. Comparison of prefix tree search to UDFs (log-scale).

searches with one or no error perform several times better than `agrep`, but as they do not apply any filtering techniques, `agrep` outperforms their implementation for larger k. Schallehn et al. [15] describe a prefix trie based index for similarity search, joins and group operations for Oracle DB. The authors introduce operators, all based on depth-first traversal, for duplicate detection in heterogenous integration scenarios that outperform non-indexed similarity operators. They did not consider any pruning. Furthermore, prefix trees are generated on-the-fly, while PETER computes them only once.

The filtering techniqes we use are contained in several other algorithms. Gravano et al. [4] introduced an efficient similarity join algorithm that uses a q-gram index to preselect similar strings. We could not compare PETER with this algorithm as there is no implementation available. The benefit of using positional q-grams was shown in [5]. As the authors implemented a sampling-based approximation for similarity string joins, one cannot directly compare PETER to their tool. Furthermore, the system was specifically designed for use in Microsoft SQL Server, whereas PETER is built on top of another RDBMS.

Xiao et al. [21] proposed to analyze mismatching q-grams for candidate preselection in near-duplicate detection. They derived two new lower bounds for edit distance, one of which we also use for suffix filtering. A direct comparison between PETER and this method would be difficult, as it directly targets duplicate detection within one relation, while we have a far more general data structure.

Aghili et al. [1] use frequency filtering as part of a vector transformation. They apply discrete wavelet and discrete fourier transformation for pre-filtering approximate join candidates in biological databases. We could not compare PETER with Aghili et al. since no implementation is available.

7 Conclusion

We presented PETER, an efficient data structure for similarity search and similarity joins on large sets of long strings. We showed that PETER outperforms all other methods we compared to, either inside or outside a RDBMS, in all performed similarity operations. Interestingly, it also outperforms exact joins inside a RDBMS.

References

1. S. A. Aghili, D. Agrawal, and A. E. Abbadi. Bft: Bit filtration technique for approximate string join in biological databases. In *String Processing and Information Retrieval*. Springer, 2003.
2. S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17), September 1997.
3. J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12, 1984.
4. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB 2001*.
5. L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *WWW 2003*.
6. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, May 1997.
7. J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A Frequent-Pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1), 2004.
8. R. Jampani and V. Pudi. Using Prefix-Trees for efficiently computing set joins. In *DASFAA 2005*.
9. A. Kalyanaraman and S. Alaru. Expressed sequence tags: Clustering and applications. In *Handbook of Computational Molecular Biology*, Boca Raton, 2006. Chapman & Hall / CRC computer information science.
10. N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB 2004*.
11. T. Lee, Y. Pouliot, V. Wagner, P. Gupta, D. S. Calvert, J. Tenenbaum, and P. Karp. Biowarehouse: a bioinformatics database warehouse toolkit. *BMC Bioinformatics*, 7, 2006.
12. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4), 1968.
13. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1), 2001.
14. NCBI. dbEST. http://www.ncbi.nlm.nih.gov/dbest, 1992.
15. E. Schallehn, K.-U. Sattler, and G. Saake. Efficient similarity-based operations for data integration. *Data & Knowledge Engineering*, 48, 2004.
16. H. Shang and T. Merrett. Tries for approximate string matching. *IEEE TKDE*, 8(4), 1996.
17. E. Sutinen and J. Tarhio. Filtration with q-Samples in approximate string matching. In *CPM 1996*.
18. E. Sutinen and J. Tarhio. On using q-Gram locations in approximate string matching. In *ESA 1995*. Springer-Verlag.
19. R. A. Wagner and M. J. Fischer. The String-to-String correction problem. *Journal of the ACM (JACM)*, 21(1), 1974.
20. H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE TKDE*, 14(1), 2002.
21. C. Xiao, W. Wang, and X. Lin. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. In *VLDB 2008*.