Efficient Similarity Search in Very Large String Sets

Dandy Fenz[1], Dustin Lange[1], Astrid Rheinländer[2], Felix Naumann[1], and Ulf Leser[2]

[1] Hasso Plattner Institute, Potsdam, Germany
[2] Humboldt-Universität zu Berlin, Department of Computer Science, Berlin, Germany

Abstract. String similarity search is required by many real-life applications, such as spell checking, data cleansing, fuzzy keyword search, or comparison of DNA sequences. Given a very large string set and a query string, the string similarity search problem is to efficiently find all strings in the string set that are similar to the query string. Similarity is defined using a similarity (or distance) measure, such as edit distance or Hamming distance. In this paper, we introduce the State Set Index (SSI) as an efficient solution for this search problem.
SSI is based on a trie (prefix index) that is interpreted as a nondeterministic finite automaton. SSI implements a novel state labeling strategy making the index highly space-efficient. Furthermore, SSI's space consumption can be gracefully traded against search time.
We evaluated SSI on different sets of person names with up to 170 million strings from a social network and compared it to other state-of-the-art methods. We show that in the majority of cases, SSI is significantly faster than other tools and requires less index space.

1 Introduction

Many applications require error-tolerant string search. For example, consider a search application for customer support in a company. While search queries may contain an incorrect spelling of a name, the search application should nevertheless find the matching entry of the customer in the database. Another application arises in a biomedical context. To find and compare genomic regions in the human genome, search applications need to account for individual variations or mutations in the genes. In these and many other scenarios, the data set consists of millions of strings, while the search application is required to answer similarity queries in subseconds.

In this paper, we tackle the *string similarity search problem*, which returns for a given query all strings from a given bag of strings that are similar to the query with respect to a previously defined string distance measure and a given distance threshold. This problem has been covered since the 1960s [26], and is also known as approximate string matching [19], string proximity search [24], or error-tolerant search [4]. Much effort has been spent by the research community

to develop filtering techniques, indexing strategies, or fast string similarity algorithms that improve the query execution time of similarity-based string searches. However, a fast query execution is often accomplished by storing a wealth of information in huge indexes in main memory. For very large string collections with hundreds of millions of strings, this approach often fails since indexes grow too large.

We propose the State Set Index (SSI) as a solution for this problem. The main advantage of SSI is that it has a very small memory footprint while providing fast query execution times on small distance thresholds at the same time. In particular, we extend and improve TITAN [15], a trie index that is interpreted as a nondeterministic finite automaton (NFA), developed by Liu et al. The contributions of this paper are:

- We introduce a novel state labeling approach, where only information on the existence of states is stored. Different from previous approaches, state transitions do not need to be stored and can be calculated on-the-fly.
- Using this highly space-efficient labeling strategy, SSI is capable of indexing very large string sets with low memory consumption on commodity hardware.
- SSI allows a graceful trade-off between index size and search performance by parameter adjustment. These parameters, namely labeling alphabet size and index length, determine the trade-off between index size and query runtime. We comprehensively evaluate these parameter settings and derive favorable settings such that index sizes remain small and the search performance is still competitive.
- We evaluate SSI on several data sets of person names coming from a social network. Our evaluation reveals that SSI outperforms other state-of-the-art approaches in the majority of cases in terms of index size and query response time. In particular, on a data set with more than 170 million strings and a distance threshold of 1, SSI outperforms all other methods we compared to.

The remainder of this paper is structured as follows: Section 2 describes related work. In Sec. 3, we cover basic definitions that are necessary for the following sections. We describe our approach in Sec. 4 by defining the index structure and algorithms for building the index and searching with it. We show evaluation results in Sec. 5 and conclude the paper in Sec. 6.

2 Related Work

In the past years, the research community has spent much effort on accelerating similarity-based string matching. Prominent approaches use prefiltering techniques, indices, refined algorithms for computing string similarity, or all in combination [19, 20].

Filter methods are known to reduce the search space early using significantly less computational effort than computing the edit distance (or another similarity measure) directly. As a result of this so-called *filter-and-verify* approach [28], only

a few candidate string pairs need to be compared using edit distance. Prominent pre-filtering approaches are based on q-grams [8, 9, 14], character frequencies [1], or length filtering [2].

Tries as an index structure for strings and exact string matching in tries were first introduced by Morrison [16] and were later extended by Shang et al. [25] with pruning and dynamic programming techniques to enable similarity-based string matching. Next to similarity-based string searches, tries and trie-based NFAs are also known to perform well in other areas, such as exact pattern matching [10], set joins [12], or frequent item set mining [7, 11].

The Peter index structure [22] was designed for near-duplicate detection in DNA data and combines tries with filtering techniques to enable similarity-based string searches and joins. It stores additional information at each trie node for early search space pruning. Pearl [23] is a follow-up where restrictions on small alphabets were removed and a strategy for parallelizing similarity searches and joins was introduced. Closely related to SSI is TITAN [15], an index structure based on prefix trees that are converted into non-deterministic automata A, such that the initial state of A corresponds to the root node of the originating prefix tree, and leaf nodes correspond to accept states in A. Additionally, further state transitions are introduced in order to enable delete, insert, and replacement operations on edit distance-based queries.

Algorithms based on neighborhood generation were first used for similarity-based string matching by Myers [17]. One drawback of the original algorithm by Myers is its space requirements, that makes it feasible only for small distance thresholds and small alphabets. The FastSS index [3] captures neighborhood relations by recursively deleting individual characters and reduces space requirements by creating a so-called k-deletion neighborhood. Similar to filtering approaches, FastSS performs search space restriction by analyzing the k-deletion neighborhood of two strings. By adding partitioning and prefix pruning, Wang et al. [27] significantly improved the runtime of similarity search algorithms based on neighborhood generation.

The Flamingo package [2] provides an inverted-list index that is enriched with a charsum and a length filter. The filter techniques are organized in a tree structure, where each level corresponds to one filter.

We empirically compare the State Set Index to FastSS, Flamingo, Pearl, and TITAN, and show that it often outperforms these tools both in terms of query execution time and with respect to index size (see Sec. 5). We could not compare to Wang et al. since no reference implementation was available.

3 Basic Concepts and Definitions

In this section, we define basic terms and concepts that we use in the subsequent sections.

3.1 Similarity Search and Measures

Let Σ be an alphabet. Let s be a string in Σ^*. 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*, $s[i \dots |s|]$ *suffix* and $s[i \dots j], (1 \leq i \leq j \leq |s|)$, *infix* of s. Any infix of length $q \in \mathbb{N}$ is called *q-gram*. Conceptually, we ground our index structure and associated algorithms on a similarity search operator defined as follows:

Definition 1 (Similarity search). *Given a string s, a bag S of strings, a distance function d and a threshold k, the similarity search operator $sSearch(s, S)$ returns all $s_i \in S$ for which $d(s, s_i) \leq k$.*

All similarity-based search operations must be based on a specific similarity measure. Though there exist several techniques to measure the similarity of two strings, we focus on *edit distance* for the scope of this paper.

Definition 2 (Edit distance [13]). *The edit distance $d_{ed}(s_1, s_2)$ of two strings s_1, s_2 is the minimal number of insertions, deletions, or replacements of single characters needed to transform s_1 into s_2. Two strings are within edit distance k, if and only if $d_{ed}(s_1, s_2) \leq k$.*

Variations of edit distance apply different costs for the three edit operations. While SSI is applicable to all edit distance-based measures with integer costs for the different edit operations, we only consider the standard definition with equal weights in this paper.

The edit distance $d_{ed}(s_1, s_2)$ can be computed by dynamic programming in $\Theta(|s_1| * |s_2|)$. Apart from the dynamic programming algorithm with quadratic complexity in time and space, there exist various improvements for the edit distance computation. Bit-parallel algorithms [18] achieve a complexity of $O(\frac{|s_1|*|s_2|}{w})$, where w is the size of the computer word. If a maximum allowed distance threshold is defined in advance, the k-banded alignment algorithm [5] computes the edit distance of two strings in $\Theta(k \cdot max\{|s1|, |s2|\})$.

3.2 Tries and NFAs

The construction of *prefix or suffix trees* is a common technique for string search. In the literature, such a tree is also called *trie* [6].

Definition 3 (Trie [6]). *A trie is a tree structure (V, E, v_r, Σ, L), where V is the set of nodes, E is the set of edges, v_r is the root node, Σ is the alphabet, and $L : V \to \Sigma^*$ is the labeling function that assigns strings to nodes. For every node v_c with $L(v_c) = s[1 \dots n]$ that is a child node of v_p, it holds $L(v_p) = s[1 \dots n-1]$, i.e., any parent node is labeled with the prefix of its children.*

The trie root represents the empty string. The descendants of a node represent strings with a common prefix and an additional symbol from the alphabet. A trie is processed from the root to the leaf nodes. Indexed strings are attached

to the node that can be reached by processing the complete string. Tries are an efficient method for exact string search.

For efficient similarity search, a trie can also be interpreted as a *nondeterministic finite automaton* [15].

Definition 4 (Nondeterministic finite automaton (NFA) [21]). *A nondeterministic finite automaton is defined as a tuple* $(Q, \Sigma, \delta, q_0, F)$*, where* Q *is the set of states,* Σ *is the input alphabet,* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ *is the state transition function (with* ε *referring to the empty word),* q_0 *is the start state, and* F *is the set of accepting states.*

The NFA begins processing in the start state q_0. The input is processed character-wise with the state transition function. An NFA is allowed to have several active states at the same time. If, after processing the entire string, the NFA is in at least one accepting state, the string is accepted, otherwise rejected.

For similarity search, we interpret a trie as an NFA. In the NFA version of the trie, the trie root node is the start state of the NFA. The nodes with associated result strings are marked as accepting states. The trie's edges are interpreted as state transitions with reading symbols. In addition, the NFA version contains for each state transition one additional state transition for reading ε as well as one ε-transition from each state to itself. These ε-transitions allow state transitions that simulate deletion, insertion, and replacement of symbols as necessary for edit distance calculation. To do similarity search with the NFA, the query string is processed as input symbol sequence by the NFA. After the processing step, the NFA is in zero, one, or more accepting states. The query result contains all strings that are attached to the reached accepting states.

The NFA idea described so far generates for a large amount of indexed strings a large automaton with many states (but there can be no false positives in the result string set). In the next section, we describe our approach that restricts the number of NFA states and checks the result string set for false positives.

4 State Set Index

The State Set Index (SSI) is an efficient and configurable index structure for similarity search in very large string sets. In this section, we first describe the key ideas of SSI before giving details on the indexing and searching algorithms.

4.1 Index Structure

SSI is based on a trie that is interpreted as an NFA. In the following, we describe the key ideas behind SSI that go beyond the basic trie and NFA concepts described above.

State labeling. The SSI states are labeled with numbers. Each label is calculated from the history of read symbols. For this purpose, the original input alphabet, in the following referred to as Σ_I, is mapped to a labeling alphabet $\Sigma_L = \{1, 2, \ldots, c_{max}\} \subset \mathbb{N}$ with $c_{max} \leq |\Sigma_I|$. A mapping function $m : \Sigma_I \rightarrow \Sigma_L$ defines the mapping of characters from the two alphabets. A label for a state with read symbols $s_1 \ldots s_{n-1} s_n \in \Sigma_I^n$ can be calculated as follows:

$$l(s_1 \ldots s_{n-1} s_n) = l(s_1 \ldots s_{n-1}) \cdot |\Sigma_L| + m(s_n)$$
$$l(\varepsilon) = 0$$

with ε referring to the empty word.

Restriction of labeling alphabet size. SSI allows to restrict the size of the labeling alphabet Σ_L. When choosing a labeling alphabet with $c_{max} < |\Sigma_I|$ (note the strict "less than" sign), at least two symbols from the input alphabet are mapped to the same symbol from the labeling alphabet.

This can reduce the number of existing states in the resulting NFA. For any two prefixes p_1, p_2 of two indexed strings, the states $l(p_1), l(p_2)$ are merged iff $l(p_1) = l(p_2)$. This is the case iff for at least one character position pos in p_1 and p_2, it holds $p_1[1 : pos - 1] = p_2[1 : pos - 1]$ and $p_1[pos] \neq p_2[pos]$ and $m(p_1[pos]) = m(p_2[pos])$, i.e., two different characters at the same position are mapped to the same symbol in the labeling alphabet and the prefixes of the strings before this character match.

Depending on the chosen mapping, a state may contain several different strings. With $c_{max} < |\Sigma_I|$, it is not possible to reconstruct a string from a state label, as there are several different possibilities for that. Thus, we need to store which strings are stored at which state. In addition, it is possible that the accepting states may contain false positives, i.e., strings that are not query-relevant. This makes it necessary to check all resulting strings by calculating the exact distance to the query string before returning results.

Choosing a labeling alphabet size is thus an important parameter for tuning the SSI. A too large labeling alphabet size results in a large NFA with many states and thus large storage requirement, but few false positives. In contrast, a too small alphabet size leads to an NFA with only few states that does not restrict the number of result strings enough; the consequence is a large number of false positives.

Restriction of index length. SSI allows to restrict the number of indexed symbols. From each string $s_1 \ldots s_n \in \Sigma_I^n$, only a prefix with a maximum length of ind_{max} is indexed. The "leaf" states contain all strings with a common prefix.

Restricting the index length can reduce the number of existing states. For any two strings g_1, g_2, the two states $l(g_1)$ and $l(g_2)$ are equal iff $l(g_1[1 : ind_{max}]) = l(g_2[1 : ind_{max}])$.

Similar to choosing the labeling alphabet size, we face the challenge of handling possible false positives in the result string sets also for restricted index

length. The index length is thus a second parameter to tune the trade-off between a large index (large index length, high memory consumption) and a large number of false positives to be handled (small index length, low memory consumption). In our analysis of a large string data set with the Latin alphabet (and some special characters) as input alphabet, we observed optimal results with a labeling alphabet size of 4 and an index length of 14 (see Sec. 5.1 for a discussion of this experiment).

Restricting the labeling alphabet size as well as the index length can significantly decrease the number of existing states. For example, in a data set with 1 million names, the mapping from the Latin alphabet to a labeling alphabet with 4 and restricting the index length to 14 results in a state count reduction from 5,958,916 states to 2,298,209 states (a reduction ratio of 61 %).

Storing states. Due to the history-preserving state labels, all potential successors of a state can be calculated. With a calculated state label, it is easy to check whether such a state exists: For any state ϕ, the state transition with the character $c \in \Sigma_L$ by definition exists iff $\phi_c = \phi \cdot |\Sigma_L| + c$ exists. This is because for any character $c' \in \Sigma_L \setminus \{c\}$, it holds $c \neq c'$ and thus $\phi_{c'} = \phi \cdot |\Sigma_L| + c' \neq \phi_c$.

To benefit from this observation, SSI only stores which states that actually exist, i.e., only states ϕ for which there is at least one prefix p of a string in the indexed string data set with $l(p) = \phi$. This reduces the necessary storage capacity, because it is not necessary to store state transitions. Also, during query answering, checking the existence of state transitions is not required.

Because SSI state labels are numbers, a simple storage format can be defined. A bitmap, where each bit combination represents a label of an existing or non-existing state, is sufficient to store which states do exist.

Storing data. Due to the introduced restrictions, an accepting state may refer to multiple, different strings – the strings cannot be completely reproduced from the state labels. Thus, it is necessary to store the strings behind the states. The required data store has a rather simple interface: A set of keys (the accepting states), each with a set of values (the strings referred to by the states) needs to be stored. Any key/multi-value store is suitable for this task. Since the data store is decoupled from the state store, the data store can be held separately. Thus, while the state store can be configured to be small or large enough to fit into main memory, the data store can be held in secondary memory.

4.2 Algorithms

In the following, we describe the details for indexing a large string set with SSI and searching with the created index.

Indexing. The indexing process is shown in Algorithm 1. All strings to be indexed are processed one after another. Each string is read character-by-character.

Algorithm 1 Indexing with SSI

Input: set of strings to be indexed $stringSet$,
labeling alphabet size c_{max},
index length ind_{max},
mapping function $m : \Sigma_I \rightarrow \Sigma_L$
Output: set of existing states $stateSet$,
set of accepting states $acceptingStateSet$,
map of states with indexed strings $dataStore$

```
1: stateSet := {}
2: acceptingStateSet := {}
3: dataStore = {}
4: for all str ∈ stringSet do
5:     state := 0
6:     for pos := 1 → min(ind_max, |str|) do
7:         state := state · c_max + m(str[pos])
8:         stateSet.add(state)
9:     acceptingStateSet.add(state)
10:    dataStore.add(state, str)
11: return stateSet, acceptingStateSet, dataStore
```

After reading a character, the current state is calculated and stored. Finally, after reading the entire string, the last state is marked as accepting state and the string is stored at this state's entry in the data store. After the initial indexing process, it is also possible to index additional strings using the same steps.

Example. We illustrate the SSI index with an example. Consider the strings Müller, Mueller, Muentner, Muster, and Mustermann and the alphabet mapping shown in Table 1. In this example, we chose a labeling alphabet size of $c_{max} = 4$ and an index length of $ind_{max} = 6$.

Σ_L	M	u	e	l	r	ü	n	t	s	m	a
Σ_I	1	2	3	4	1	2	3	4	1	2	3

Table 1. Example for alphabet mapping function $m : \Sigma_L \rightarrow \Sigma_I$

Figure 1 shows all existing states of the resulting index. The accepting states point to the indexed strings as follows:
1869 → {Müller}, 1811→ {Mueller}, 1795→ {Muenter}, 1677 → {Muster, Mustermann} □

Searching. We now describe how to process a query string $q \in \Sigma_I^*$ with edit distance $k \in \mathbb{N}$. The search process is shown in Algorithm 2.

First, a set S of cost-annotated states s with state ϕ_s and associated costs λ_s (the number of edit distance operations required so far) is created. We write

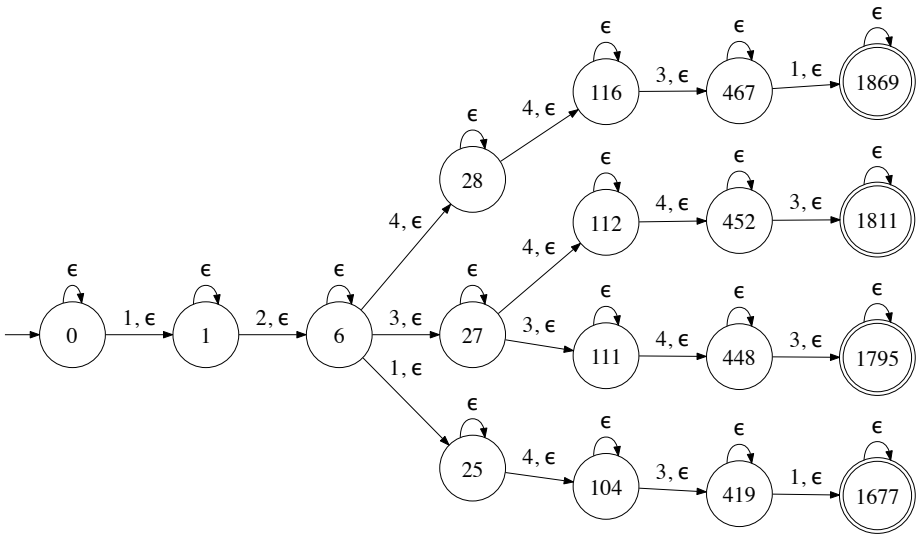


Fig. 1. Example for states created by SSI with $c_{max} = 4$ and $ind_{max} = 6$

$s := \langle \phi_s, \lambda_s \rangle$. Initially, all states that can be reached from the start state with at most k ε-transitions are added to S. To determine these states, the labels of the successors of the start state are calculated and their existence is validated. If a state s in S is associated with several different costs λ_s, only the record with the lowest λ_s is kept; all other records are dismissed. This selection is done for all state calculations and is not stated again in the following.

Next, the query string q_I is translated into the labeling alphabet with $q := l(q_I)$. The characters of q are processed one-by-one. The following steps are processed for each character c in q.

Another empty set S^* of current cost-annotated states is created. For each cost-annotated state $\langle \phi_s, \lambda_s \rangle$ in S, a set S_c^* is created and processed with the following steps:

- To simulate deletion of characters, the cost-annotated state $\langle \phi_s, \lambda_s + 1 \rangle$ is added to S_c^* if $\lambda_s + 1 \leq k$.
- To simulate matching of characters, it is checked whether the state $\phi_s^* := \phi_s \cdot |\Sigma_L| + i$ exists. This state exists if and only if there is a transition from ϕ_s with the character c to ϕ_s^*. If the state exists, then $\langle \phi_s^*, \lambda_s \rangle$ is added to S_c^*.
- Next, the insertion of characters other than c is simulated. If $\lambda_s + 1 \leq k$, then for each $\phi_s^* := \phi_s \cdot |\Sigma_L| + m(c^*)$ with $c^* \in \Sigma_L \setminus \{c\}$, a new cost-annotated state $\langle \phi_s^*, \lambda_s + 1 \rangle$ is added to S_c^*.
- Inserting characters is simulated using ε-transitions. For each cost-annotated state $\langle \phi_s, \lambda_s \rangle$ in S_c^*, all states ϕ_s^* are determined that can be reached from ϕ_s with k ε-transitions. For each such state, the annotated states $\langle \phi_s^*, \lambda_s + i \rangle$ with $\lambda_s \leq i \leq k$ are added to S_c^*.

Algorithm 2 Searching with SSI

Input: query string q, maximum edit distance k
Output: result string set R

1: $S := \{\langle i \cdot c, i\rangle \mid 0 \leq i \leq k, c \in \Sigma_L\} \cap stateSet$ Initial ε-transitions
2: **for** $pos := 1 \rightarrow min(ind_{max}, |q|)$ **do**
3: $S^* := \{\}$
4: **for all** $\langle \phi_s, \lambda_s\rangle \in S$ **do**
5: $S_c^* := \{\}$
6: **if** $\lambda_s + 1 \leq k$ **then** Deletion
7: $S_c^* := S_c^* \cup \langle \phi_s, \lambda_s + 1\rangle$
8: **for** $i := 1 \rightarrow |\Sigma_L|$ **do** Match & Substitution
9: $\phi_s^* := \phi_s \cdot |\Sigma_L| + i$
10: **if** $\phi_s^* \in stateSet$ **then**
11: **if** $i = m(q[pos])$ **then**
12: $S_c^* := S_c^* \cup \langle \phi_s^*, \lambda_s\rangle$
13: **else if** $\lambda_s + 1 \leq k$ **then**
14: $S_c^* := S_c^* \cup \langle \phi_s^*, \lambda_s + 1\rangle$
15: $S_c^* := S_c^* \cup (\{\langle \phi_s + i \cdot c, i\rangle \mid s \in S_c^*, \lambda_s \leq i \leq k, c \in \Sigma_L\} \cap stateSet)$ Insertion
16: $S^* := S^* \cup S_c^*$
17: $S := S^*$
18: $R := \{\}$ Retrieve strings and filter by distance
19: **for all** $\langle \phi_s, \lambda_s\rangle \in S$ **do**
20: **if** $\phi_s \in acceptingStateSet$ **then**
21: $R := R \cup \{s \in dataStore.get(\phi_s) \mid d_{ed}(s, q) \leq k\}$
22: **return** R

Then, S is replaced by S^* and all steps are repeated with the next character. After processing all characters, the state set S represents the final state set. For all states from S that are accepting, all strings stored at those states are retrieved. This set of strings is filtered by calculating the actual edit distance to the query string as it may contain false positives. This set of filtered strings is the result of the search.

Example. Consider the index in Fig. 1 and the example query Mustre with a maximum distance of $k = 2$. The initial state set $S = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 6, 2\rangle\}$ contains all states reachable from the start state with at most $k = 2$ ε-transitions. Next, the first character $c = 1$ (M) is processed. The state sets $S^* = S_c^* = \emptyset$ are created. For all entries in S, the five above-described steps are executed. After processing the first character, we have:

$$S = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 6, 1\rangle, \langle 28, 2\rangle, \langle 27, 2\rangle, \langle 25, 2\rangle\}$$

After that, the character $c = 2$ (u) is processed. The state set after this step is:

$$S = \{\langle 0, 2\rangle, \langle 1, 1\rangle, \langle 6, 0\rangle, \langle 28, 1\rangle, \langle 27, 1\rangle, \langle 25, 1\rangle, \langle 116, 2\rangle, \langle 112, 2\rangle, \langle 111, 2\rangle, \langle 104, 2\rangle\}$$

After processing the third character $c = 1$ (s), we have:

$$S = \{\langle 1, 2\rangle, \langle 6, 1\rangle, \langle 28, 1\rangle, \langle 27, 1\rangle, \langle 25, 0\rangle, \langle 116, 2\rangle, \langle 112, 2\rangle, \langle 111, 2\rangle, \langle 104, 1\rangle, \langle 419, 2\rangle\}$$

The next character $c = 4$ (t) results in:

$$S = \{\langle 6, 2\rangle, \langle 28, 1\rangle, \langle 27, 2\rangle, \langle 25, 1\rangle, \langle 104, 0\rangle, \langle 116, 1\rangle, \langle 112, 1\rangle, \langle 111, 2\rangle, \langle 419, 1\rangle, \langle 1677, 2\rangle, \langle 467, 2\rangle, \langle 452, 2\rangle\}$$

The character $c = 1$ (r) is processed as follows:

$$S = \{\langle 28, 2\rangle, \langle 25, 2\rangle, \langle 104, 1\rangle, \langle 116, 2\rangle, \langle 112, 2\rangle, \langle 419, 1\rangle, \langle 1677, 1\rangle, \langle 1869, 2\rangle, \langle 467, 2\rangle, \langle 452, 2\rangle\}$$

With the last character $c = 3$ (e), we finally have:

$$S = \{\langle 104, 2\rangle, \langle 419, 1\rangle, \langle 1677, 2\rangle, \langle 467, 2\rangle, \langle 1811, 2\rangle\}$$

From the set of states in S, only the accepting states 1677 and 1811 are further processed. The strings stored at these states are Muster, Mustermann, and Mueller. After filtering false positives, we finally have the result string set {Muster}.

□

Complexity. To index n strings with a maximum index length ind_{max}, at most ind_{max} states need to be calculated for each string. Thus, we have an indexing complexity of $\mathcal{O}(n \cdot ind_{max})$.

The most important size factor of SSI is the number of created states. For an index length ind_{max} and an indexing alphabet Σ_L, the number of possible states is $|\Sigma_L|^{ind_{max}}$. The index size depends on the chosen parameters where ind_{max} is the dominant exponential parameter.

The search algorithm of SSI mainly depends on c_{max}, ind_{max}, and the search distance k. In the first step, $k \cdot |c_{max}|$ potential states are checked. For each existing state, its successor states are created. These consist of up to one state created by deletion, c_{max} states created by match or substitution, and $k \cdot |c_{max}|$ states created by insertion of a character. This process is repeated up to ind_{max} times. Overall, we have up to $(k \cdot |c_{max}|) \cdot (1 + k \cdot |c_{max}| + k \cdot |c_{max}|)^{ind_{max}}$ steps and thus a worst-case complexity of $\mathcal{O}((k \cdot |c_{max}|)^{ind_{max}})$. Similar to the indexing process, the complexity is bound by the parameters c_{max} and ind_{max} where ind_{max} is the dominant exponential factor. By evaluating the existence of states during the search process and proceeding only with existing states, we typically can significantly decrease the number of states that are actually evaluated.

5 Evaluation

We use a set of person names crawled from the public directory of a social network website to evaluate the performance of SSI for parameter selection, index creation, and for search operations. Table 5 shows some properties of our data set. The set D_{full} contains all person names we retrieved, whereas the sets D_i consist of i randomly chosen strings taken from D_{full}. First, we evaluate the impact of different parameter configurations on the performance of SSI and then choose the best setting to compare SSI against four competitors. In particular, we compare SSI to FastSS [3], TITAN [15], Flamingo [2], and Pearl [23], which are all main memory-based tools for index-based similarity string operations (see Sec. 2 for details). For Flamingo and Pearl, we use the original implementations provided by the authors. For FastSS and TITAN, we use our own implementations of the respective algorithms. Our evaluation comprises experiments both for indexing time and space as well as experiments on exact and similarity-based search queries.

All experiments were performed on an Intel Xeon E5430 processor with 48 GB RAM available using only a single thread. For each experiment, we report the average of three runs.

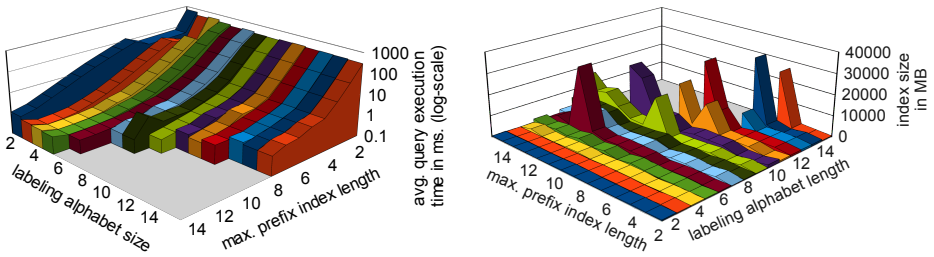
Set	# strings	avg./min./max. string length	input alphabet size	# exact duplicates
D_{full}	170,879,859	13.99 / 1 / 100	38	70,751,399
D_{200k}	200,000	14.02 / 1 / 61	29	5,462
D_{400k}	400,000	14.02 / 1 / 54	32	17,604
D_{600k}	600,000	14.01 / 1 / 55	35	35,626
D_{800k}	800,000	14.02 / 1 / 61	33	54,331
D_{1000k}	1,000,000	14.01 / 1 / 64	35	77,049

Table 2. Evaluation data sets.

5.1 Evaluation of SSI Parameters

We exemplarily used the set D_{1000k} to evaluate the impact of different parameter configurations on the performance of SSI on small string sets. Since the maximum index length ind_{max} and labeling alphabet size $|\Sigma_L|$ have a large influence on the performance of SSI, we varied both ind_{max} and $|\Sigma_L|$ in the range of 2 to 15. We could not perform experiments on larger parameter ranges due to memory constraints of our evaluation platform.

As displayed in Fig. 2(a), the average query execution time drastically decreases with increased labeling alphabet size and maximum index length. In particular, a configuration of SSI with $ind_{max} = 12$ and $|\Sigma_L| = 8$ outperforms a configuration using $ind_{max} = 2$ and $|\Sigma_L| = 2$ by three orders of magnitude (factor 1521). On the other hand, when increasing ind_{max} and $|\Sigma_L|$, we observed that the index size grows significantly (see Fig. 2(b)). For example, changing the configuration from $ind_{max} = 2$ and $|\Sigma_L| = 2$ to $ind_{max} = 6$ and $|\Sigma_L| = 13$



(a) Average query execution time (log-scale)

(b) Average index size in MBytes

Fig. 2. Evaluation of parameters ind_{max} and $|\Sigma_L|$ for D_{1000k} and $k = 1$.

increases memory requirements by a factor of 60. We also observed that the number of false positives and the number of accessed keys per query decreases both with increasing ind_{max} and $|\Sigma_L|$ size (data not shown) and conclude that this is the main reason for the positive outcome of a large labeling alphabet and a large index length. However, we could not increase both parameters further due to memory limitations of our platform, but we expect a further decrease of query execution time.

We also evaluated the influence of varying parameters on query execution time and index size on D_{full}. Results are shown in Fig. 3 for selected configurations. Similar to the experiments on D_{1000k}, the index size grows heavily while increasing ind_{max} and $|\Sigma_L|$. Particularly, choosing $|\Sigma_L| = 3$ and $ind_{max} = 15$ yields in an index size of approximately 12 GB, whereas a configuration with $|\Sigma_L| = 5$ and $ind_{max} = 15$ needs a bit vector of 28 GB. On the other hand, the query execution time decreases with elongating ind_{max} and $|\Sigma_L|$. Using $|\Sigma_L| = 3$ and $ind_{max} = 15$, the query execution time averages to 8 milliseconds, whereas with $|\Sigma_L| = 5$ and $ind_{max} = 15$ the query execution time diminishes to 2.6 milliseconds on average at the expense of a very large index. We also experimented with other settings of ind_{max} and $|\Sigma_L|$ in the range of 2 to 15, but these configurations either did not finish the indexing process in a reasonable amount of time or ran out of memory on our evaluation platform. Therefore, we did not consider these settings for parameter configuration on large string sets.

In summary, both parameter variations of $|\Sigma_L|$ and ind_{max} have a large impact on the performance of SSI. While increasing $|\Sigma_L|$ or ind_{max}, the number of false positive results that need to be verified decreases, which yields in a considerably fast query response time. However, our experiments also revealed that at some point, no further improvements on query response time can be achieved by increasing $|\Sigma_L|$ and ind_{max}. This is caused by an increased effort for calculating involved final states that outweighs the decreased amount of false-positive and the number of lookups in this setting.

Therefore, a beneficial configuration for indexing up to one million person names is to fix $|\Sigma_L| = 8$ and $ind_{max} = 12$. Using this configuration leads to a

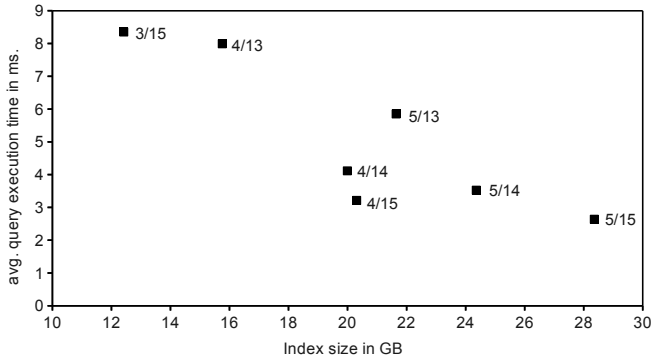


Fig. 3. Trade-off between index size and query execution time on D_{full} and $k = 1$ on varying configurations of $ind_{max} \in \{3, 4, 5\}$ and $|\Sigma_L| \in \{13, 14, 15\}$.

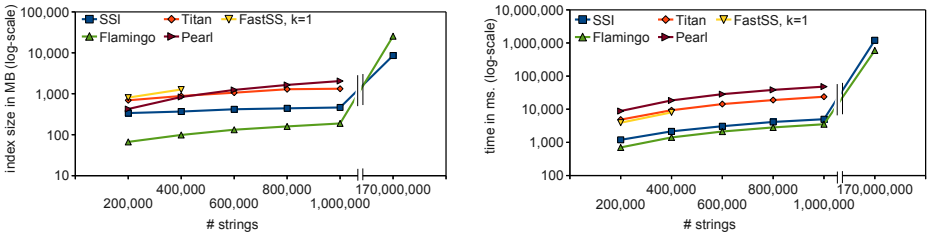
fast query execution time with on a moderate index size. When indexing larger string collections, a shift in favor of index length is reasonable, since an increasing length yields larger performance enhancements with respect to query response time. However, elongating the index length and the labeling alphabet yields also in a vast growth of the index size, but we strive for an index structure that is efficient both in terms of space and time. Thus, we decided to configure SSI with $|\Sigma_L| = 4$ and $ind_{max} = 14$ for all following experiments using D_{full}, since this configuration gives us the best query execution time with an index size of at most 20 GB.

5.2 Index Creation Time and Memory Consumption

We evaluated SSI in terms of index creation time and memory consumption and compared it to other main-memory index structures, namely FastSS, TITAN, Pearl, and Flamingo on all available data sets. For all evaluated tools, we observed that both index sizes and indexing time grow at the same scale as the data sets.

Many of the tools we compared to are not able to handle very large string collections. Figure 4(a) displays the memory consumption of each created index in main memory. We were able to index D_{full} only with SSI and Flamingo; FastSS, Pearl, and TITAN ran out of memory during index creation. In particular, FastSS even failed to create indexes with more than 400,000 strings. Another severe drawback of FastSS is that it needs to create a separate index for each edit distance threshold k – in contrast to all other evaluated tools.

Clearly, SSI outperforms all other trie- or NFA-based tools in terms of memory consumption and outperforms FastSS, Pearl, and TITAN with factors in the range of 1.4 (Pearl on D_{200k}) to 4.5 (Pearl on D_{1000k}). Compared to Flamingo, which is based on indexing strings by their lengths and char-sums, we observed that SSI is advantageous for indexing large data sets. When indexing D_{full}, SSI needs 3.0 times less memory than Flamingo. For small data sets with up to



(a) Average index size in MBytes (log-scale)

(b) Average index creation time in milliseconds (log-scale)

Fig. 4. Index creation

one million strings, Flamingo outperforms SSI with factors in the range of 2.4 (D_{1000k}) to 5.0 (D_{200k}).

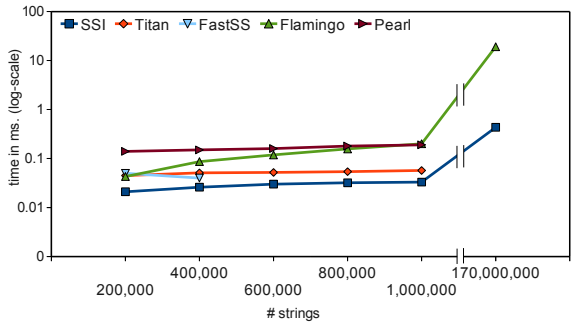
We also evaluated SSI on the time spent for index creation. As displayed in Fig. 4(b), SSI indexes all data sets significantly faster than the other trie- or NFA-based methods. It outperforms FastSS with factors 3.2 to 3.7 on $k = 1$, TITAN with factors 4.0 to 4.7, and Pearl with factors 7.4 to 9.6. Similar to the memory consumption, SSI is the more superior the larger the data sets grow. Compared to Flamingo, SSI is only slightly slower (factors in the range of 1.4 to 2.0).

5.3 Query Answering

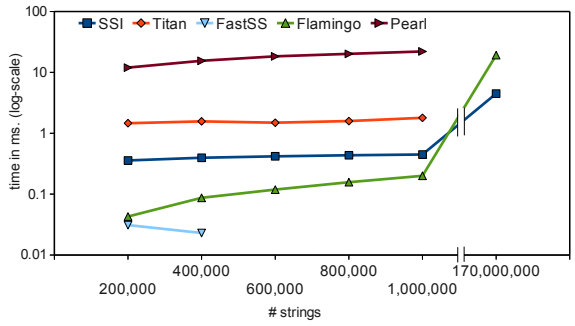
To evaluate the performance of SSI in query answering, we assembled a set of 1,000 example queries separately for each data set as follows: First, we randomly selected 950 strings from the respective data set and kept 500 of these strings unchanged. On the remaining 450 strings, we introduced errors by randomly changing or deleting one character per string. Additionally, we generated 50 random strings and added them to the set of queries. For each query, we measured the execution time and report the average of all 1,000 queries. We compared SSI to all above-mentioned tools both for exact and similarity-based queries with varying edit distance thresholds $k \in \{0, 1, 2, 3\}$. For all search experiments, indexing was performed in advance and is not included in the measured times.

For exact queries, SSI outperformed all competitors independent of the data set size (see Fig. 5(a)). Specifically, SSI outperformed FastSS with factor 2.3 on D_{200k} and factor 1.5 on D_{400k}, TITAN with factors varying between 1.6 on D_{800k} and 2.1 on D_{200k}, Pearl with factors varying between 5.3 on D_{600k} and 6.6 on D_{200k}, and Flamingo with factors from 2.0 on D_{200k} to 44.1 on D_{full}.

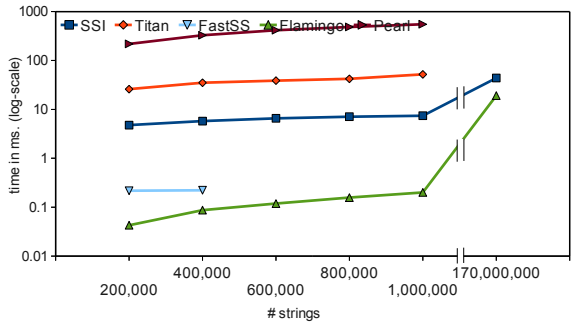
As displayed in Fig. 5(b – d), SSI significantly outperforms the trie- and NFA-based tools TITAN and Pearl on edit distance based queries. Using an edit distance threshold of $k = 1$, SSI outperforms TITAN with a factor of 4, using $k = 3$, SSI is 5.4 to 7.4 times faster than TITAN depending on the data set. Compared to Pearl, SSI is faster by more than one order of magnitude, indepen-



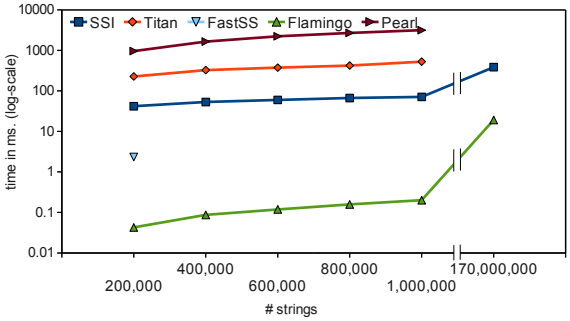
(a) k=0



(b) k=1



(c) k=2



(d) k=3

Fig. 5. Average query execution time in milliseconds (log-scale).

dent of the data set and the edit distance thresholds. However, on comparatively small data sets (D_{200k}, D_{400k}), FastSS is by an order of magnitude faster than SSI. This observation needs to be put into perspective, since FastSS on the one hand needs to create a separate index for each k, and creating indexes with more than $400,000$ strings was not possible using FastSS. In contrast, SSI does not have these limitations.

Furthermore, we acknowledge that Flamingo, which has a different indexing and search approach (cf. Sec. 2), is significantly faster than SSI in many situations. For searches in D_{full} with $k = 0$ and $k = 1$, SSI was faster by a factor of 4.2, in all other situations, Flamingo outperformed SSI. Recall that Flamingo uses considerably more memory than SSI for indexing D_{full} to achieve this (cf. Fig. 4(a)). We also clearly observe that the advantages of Flamingo grow the larger edit distance thresholds get. However, future improvements of SSI could directly address this issue, e.g., by integrating bit-parallel edit distance computation methods which provide a fast edit distance computation that is independent of the chosen threshold k.

6 Conclusion

In this paper, we presented the State Set Index (SSI), a solution for fast similarity search in very large string sets. By configuring SSI's parameters, we can scale the index size allowing best search performance given memory requirements. Our experiments on a very large real-world string data set showed that SSI significantly outperforms current state-of-the-art approaches for string similarity search with small distance thresholds.

References

1. S. A. Aghili, D. Agrawal, and A. E. Abbadi. BFT: Bit filtration technique for approximate string join in biological databases. In *Proc. of the Intl. Symposium on String Processing and Information Retrieval (SPIRE)*, pages 326–340, 2003.
2. A. Behm, R. Vernica, S. Alsubaiee, S. Ji, J. Lu, L. Jin, Y. Lu, and C. Li. UCI Flamingo Package 4.0, 2011.
3. T. Bocek, E. Hunt, and B. Stiller. Fast Similarity Search in Large Dictionaries. Technical report, Department of Informatics, University of Zurich, 2007.
4. M. Celikik and H. Bast. Fast error-tolerant search on very large texts. In *Proc. of the ACM Symposium on Applied Computing (SAC)*, pages 1724–1731, 2009.
5. J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179, 1984.
6. E. Fredkin. Trie memory. *Commun. of the ACM*, 3:490–499, September 1960.
7. G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. of the ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.
8. 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 *Proc. of the Intl. Conf. on Very Large Databases (VLDB)*, pages 491–500. Morgan Kaufmann, 2001.

9. L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *Proc. of the Intl. World Wide Web Conf. (WWW)*, pages 90–101, 2003.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
11. 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.
12. R. Jampani and V. Pudi. Using Prefix-Trees for efficiently computing set joins. In *Proc. of the Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, 2005.
13. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.
14. C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proc. of the Intl. Conf. on Data Engineering (ICDE)*, pages 257–266. IEEE Computer Society, 2008.
15. X. Liu, G. Li, J. Feng, and L. Zhou. Effective indices for efficient approximate string search and similarity join. In *Proc. of the Intl. Conf. on Web-Age Information Management*, pages 127–134. IEEE Computer Society, 2008.
16. D. R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
17. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.
18. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
19. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1), 2001.
20. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.
21. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3:114–125, April 1959.
22. A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *Proc. of the Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 519–536. Springer, 2010.
23. A. Rheinländer and U. Leser. Scalable sequence similarity search in main memory on multicores. In *International Workshop on High Performance in Bioinformatics and Biomedicine (HiBB)*, 2011.
24. S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Ozsoyoglu. Distance based indexing for string proximity search. In *Proc. of the Intl. Conf. on Data Engineering (ICDE)*, pages 125–136, 2003.
25. H. Shang and T. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8:540–547, 1996.
26. T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics and Systems Analysis*, 4:52–57, 1968.
27. W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 759–770, 2009.
28. C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. of the VLDB Endowment*, 1:933–944, August 2008.