Accurate Cardinality Estimation of Co-occurring Words Using Suffix Trees (Extended Version)

Jens Willkomm, Martin Schäler, and Klemens Böhm

Karlsruhe Institute of Technology (KIT), Germany
{jens.willkomm,martin.schaeler,klemens.boehm}@kit.edu

Abstract. Estimating the cost of a query plan is one of the hardest problems in query optimization. This includes cardinality estimates of string search patterns, of multi-word strings like phrases or text snippets in particular. At first sight, suffix trees address this problem. To curb the memory usage of a suffix tree, one often prunes the tree to a certain depth. But this pruning method "takes away" more information from long strings than from short ones. This problem is particularly severe with sets of long strings, the setting studied here. In this article, we propose respective pruning techniques. Our approaches remove characters with low information value. The various variants determine a character's information value in different ways, e.g., by using conditional entropy with respect to previous characters in the string. Our experiments show that, in contrast to the well-known pruned suffix tree, our technique provides significantly better estimations when the tree size is reduced by 60% or less. Due to the redundancy of natural language, our pruning techniques yield hardly any error for tree-size reductions of up to 50%.

Keywords: Query optimization · cardinality estimation · suffix tree.

1 Introduction

Query optimization and accurate cost estimation in particular continue to be fundamentally important features of modern database technology [43, 27]. While cardinality estimation for numerical attributes is relatively well understood [34], estimating the cardinality of textual attributes remains challenging [23, 9, 40]. This is particularly true when the query (1) contains regular expressions, e.g., aims to find the singular of a word and the plural, (2) searches for word chains, e.g., `high noon`, or (3) a combination of both, i.e., a regular expression involving several words. One application where this is necessary is text mining. Various text mining tasks query co-occurring words (co-occurrences), i.e., words alongside each other in a certain order [29, 30]. Such queries are dubbed *co-occurrence queries*. Co-occurrence queries are important because, in linguistic contexts, co-occurrences indicate semantic proximity or idiomatic expressions [24]. Optimizing such queries requires estimates of the cardinality of co-occurrences.

Problem Statement. To query co-occurrences, one uses search patterns like `the␣emancipation␣of␣*` or `emancipation␣*␣Catholics`. A particularity of co-occurrences is that they only exist in chains of several words (word chain), like

phrases or text snippets. This calls for cardinality estimates on a set of word chains. Compared to individual words, word chains form significantly longer strings, with a higher variance in their lengths. Our focus in this article is on the accuracy of such estimates with little memory consumption at the same time.

State-of-the-Art. One approach to index string attributes is to split the strings into trigrams, i.e., break them up into sequences of three characters [1, 23, 9]. This seems to work well to index individual words. However, the trigram approach will not reflect the connection between words that are part of a word chain. Another method to index string attributes is the suffix tree [23, 28]. Since suffix trees tend to be large, respective pruning techniques have been proposed. A common technique is to prune the tree to a maximal depth [23]. Since the size of a suffix tree depends on the length of the strings [37], other pruning methods work similarly. We refer to approaches that limit the depth of the tree as *horizontal pruning*. With horizontal pruning however, all branches are shortened to the same length. So horizontal pruning "takes away" more information from long strings than from short ones. This leads to poor estimation accuracy for long strings and more uncertainty compared to short ones.

Challenges. Designing a pruning approach for long strings faces the following challenges: First, the reduction of the tree size should be independent of the length of the strings. Second, one needs to prune, i.e., remove information, from both short and long strings to the same extent rather than only trimming long strings. Third, the pruning approach should provide a way to quantify the information loss or, even better, provide a method to correct estimation errors.

Contribution. In this work, we propose what we call *vertical pruning*. In contrast to horizontal pruning that reduces any tree branch to the same maximal height, vertical pruning aims at reducing the number of branches, rather than their length. The idea is to map several strings to the same branch of the tree, to reduce the number of branches and nodes. This reduction of tree branches makes the tree *thinner*. So we dub the result of pruning with our approach *thin suffix tree* (TST). A TST removes characters from words based on the information content of the characters. We propose different ways to determine the information content of a character based on empirical entropy and conditional entropy. Our evaluation shows that our pruning approach reduces the size of the suffix tree depending on the character distribution in natural language (rather than depending on the length of the strings). TST prunes both short and long strings to the same extent. Our evaluation also shows that TST provides significantly better cardinality estimations than a pruned suffix tree when the tree size is reduced by 60% or less. Due to the redundancy of natural language, TST yields hardly any error for tree-size reductions of up to 50%.

Paper Outline. Section 2 features related work. We introduce the thin suffix tree in Section 3. We say how to correct estimation errors in Section 4. We describe the procedures *insert* and *query* in Section 5. Our evaluation is in Section 6.

2 Related Work

This section is split into three parts. First, we summarize lossless methods to compress strings and suffix trees. These methods reduce the memory consumption without loss of quality. Such methods allow for a perfect reconstruction of the data and provide exact results. Second, we turn to pruning methods for suffix trees. Pruning is a lossy compression method that approximates the original data. Finally, we review empirical entropy in string applications.

String Compression Methods. There exist various methods to compress strings. One is statistical compression, like Huffman coding [20] and Hu-Tucker coding [19]. Second, there are compressed text self-indexes, like FM-index [13]. Third, there is dictionary-based compression, like the Lempel and Ziv (LZ) compression family [44, 42, 3]. Fourth, there are grammar-based compression methods, like Re-Pair [26]. However, these compression methods are either incompatible with pattern matching or orthogonal to our work.

Suffix Tree Compression Methods. A suffix tree (or trie) is a data structure to index text. It efficiently implements many important string operations, e.g., matching regular expressions. To reduce the memory requirements of the suffix tree, there exist approaches to compress the tree based on its structure [35]. Earlier approaches are path compression [22, 21] and level compression [2]. More recent compression methods and trie transformations are path decompositions [17], top trees [4], and hash tables [36]. Our approach in turn works on the input strings rather than on the tree structure.

In addition to structure-based compression, there exist alphabet-based compression techniques. Examples are the compressed suffix tree [18], the sparse suffix tree [25] and the idea of alphabet sampling [10, 16]. These methods reduce the tree size at the expense of the query time. All these methods provide exact results, i.e., are lossless compression methods. Lossless tree compression is applicable in addition to tree pruning, i.e., such methods work orthogonal to our approach.

Suffix Tree Pruning. Suffix trees allow to estimate the cardinality of string predicates, i.e., the number of occurrences of strings of arbitrary length [41]. With large string databases in particular, a drawback of suffix trees is their memory requirement [12, 41]. To reduce the memory requirements of the suffix tree, variants of it save space by removing some information from the tree [23].

We are aware of three approaches to select the information to be removed: A first category is data-insensitive, application-independent approaches. This includes shortening suffixes to a maximum length [23]. Second, there are data-sensitive, application-independent pruning approaches that exploit statistics and features of the data, like removing infrequent suffixes [15]. Third, there are data-sensitive, application-dependent approaches. They make assumptions on the suffixes which are important or of interest for a specific application. Based on the application, less useful suffixes are removed [38]. For example, suffixes with typos

or optical character recognition errors are less useful for most linguistic applications. It is also possible to combine different pruning approaches. In this work, we focus on data-sensitive and application-independent pruning.

Horizontal Pruning Approaches. Existing pruning techniques usually reduce the height of the tree, by pruning nodes that are deeper than a threshold depth [23]. Another perspective is that all nodes deeper than the threshold are merged into one node. We in turn propose a pruning technique that reduces the width of the tree rather than the depth.

Empirical Entropy in String Applications. The usage frequency of characters in natural language is unevenly distributed [39]. For this reason, the empirical entropy is an essential tool in text and string compression. It is used in optimal statistical coding [20] and many data compression methods [14, 3].

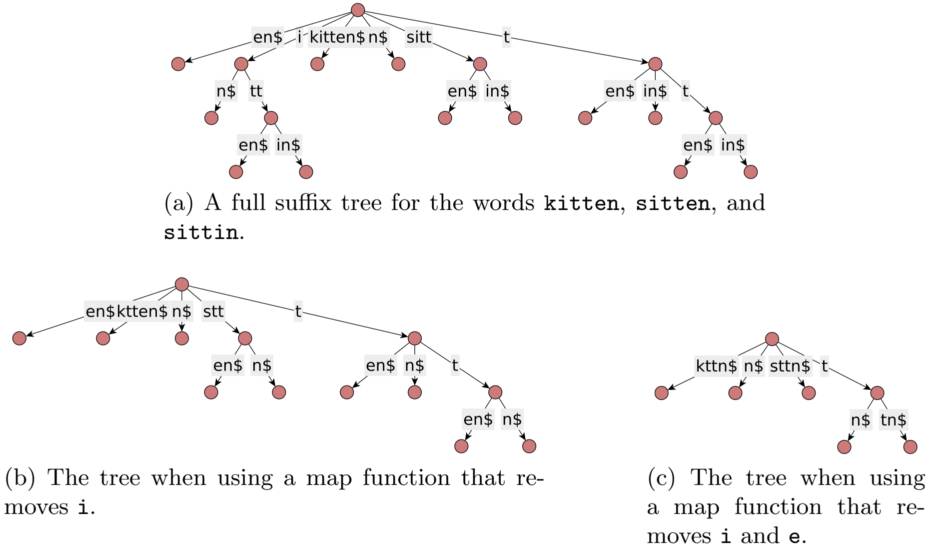
3 The Thin Suffix Tree

To store word chains as long strings in a suffix tree efficiently, we propose the thin suffix tree (TST). In contrast to horizontal pruning approaches, it aims at reducing the number of branches in the tree, rather than their length. We refer to this as *vertical pruning*. The idea is to conflate branches of the tree to reduce its memory consumption. This means that one branch stands for more than one suffix. As usual, the degree of conflation is a trade-off between memory usage and accuracy. In this section, we (1) present the specifics of TST and (2) define interesting map functions that specify which branches to conflate.

3.1 Our Vertical Pruning Approach

To realize the tree pruning, we propose a map function that discerns the input words from the strings inserted into the tree. For every input word (preimage) that one adds to the tree, the map function returns the string (image) that is actually inserted. TST stores the image of every suffix. This map function is the same for the entire tree, i.e., we apply the same function to any suffix to be inserted (or to queries). Thinning occurs when the function maps several words to the same string. The map function is surjective.

Fixing a map function affects the search conditions of the suffix tree. A suffix tree and suffix tree approximation techniques usually search for exactly the given suffix, i.e., all characters in the given order. Our approximation approach relaxes this condition to words that contain the given characters in the given order, but may additionally contain characters that the map function has removed. For example, instead of querying for the number of words that contain the exact suffix `mnt`, one queries the number of words that contain the characters `m`, `n`, and `t` in this order, i.e., a more general pattern. We implement this by using a map function that removes characters from the input strings. We see the following two advantages of such a map functions: (1) Branches of similar suffixes conflate. This reduces the number of nodes. (2) The suffix string gets shorter. This reduces the number of characters. Both features save memory usage.



(a) A full suffix tree for the words `kitten`, `sitten`, and `sittin`.

(b) The tree when using a map function that removes `i`.

(c) The tree when using a map function that removes `i` and `e`.

Fig. 1. The impact of character-removing map functions on a suffix tree.

3.2 Character-removing Map Functions

A character-removing map function removes characters from a given string.

Definition 1. *A* character-removing map function *is a function that maps a preimage string to an image string by removing a selection of specific characters.*

To remove characters systematically, we consider the information value of the characters. According to Shannon's information theory, common characters carry less information than rare ones [8, 33]. This is known as Shannon entropy of the alphabet. The occurrence probability $P(c)$ of a character c is its occurrence frequency relative to the one of all characters of the alphabet Σ.

$$P(c) = \frac{\text{frequency}(c)}{\sum_{\sigma \in \Sigma} \text{frequency}(\sigma)} \tag{1}$$

The information content of a character c is inversely proportional to its occurrence probability $P(c)$.

$$\mathrm{I}(c) = \frac{1}{P(c)} \tag{2}$$

According to Zipf's law, the occurrence probability of each c is inversely proportional to its rank in the frequency table [39].

$$P(c) \sim \frac{1}{\text{rank}(c)} \tag{3}$$

Thus, the information content of character c is proportional to its rank.

$$\mathrm{I}(c) \sim \text{rank}(c) \tag{4}$$

To create an tree of approximation level n, a map function removes the n most frequent characters in descending order of their frequency.

Example 1. The characters `e`, `t`, and `a` (in this order) are the three most frequent characters in English text. At approximation degree 3, a map function maps the input word `requirements` to the string `rquirmns`.

Example 2. Figure 1a shows the full suffix tree for the words `kitten`, `sitten`, and `sittin`. Its size is 288 bytes. The exact result of the regular expression `*itten` is 2 counts. For illustrative purposes, we first show the impact of a map function that removes character `i` and of a second function that removes `i` and `e`. Figure 1b shows the first approximation level, i.e., character `i` removed. The tree is of size 224 bytes, i.e., it saves 22% of memory used. When we query this tree for the regular expression `*itten`, the result is 2 counts. The tree still estimates correctly. Figure 1c shows a tree with character `e` removed additionally. It has a size of 192 bytes and, thus, saves 33% of memory usage. When we query the regular expression `*itten`, the result is 3 counts. So it now overestimates the cardinality.

3.3 More Complex Map Functions

When removing characters of the input words, one can also think of more complex map functions. We see two directions to develop such more complex functions: (1) Consider character chains instead of single characters or (2) respect previous characters. We will discuss both directions in the following.

Removing Character Chains. The map function considers the information content of combinations of several consecutive characters, i.e., character chains. Take a string $s = c_1c_2c_3$ that consists of characters c_1, c_2, and c_3. We consider character chains of length o and create a frequency table of character chains of this length. The information content of a character chain $c_1 \dots c_o$ is proportional to its rank.

$$\mathrm{I}(c_1 \dots c_o) \sim \mathrm{rank}(c_1 \dots c_o) \tag{5}$$

Our character-chain-removing map function is a character-removing map function that removes every occurrence of the n most frequent character chains of the English language of length o.

Example 3. A character-chain-removing map function removing the chain `re` maps the input `requirements` to `quiments`.

Using Conditional Entropy. We define a character-removing map function that respects one or more previous characters to determine the information content of a character c_1. Given a string $s = c_0c_1c_2$, instead of using the character's occurrence probability $P(c_1)$, a conditional-character removing map function

considers the conditional probability $P(c_1|c_0)$. Using Bayes' theorem, we can express the conditional probability as follows.

$$P(c_1|c_0) = \frac{\text{frequency}(c_0c_1)}{\text{frequency}(c_0)} \tag{6}$$

Since the frequencies for single characters and character chains are roughly known (for the English language for instance), we compute all possible conditional probabilities beforehand. So we can identify the n most probable characters with respect to previous characters to arrive at a tree approximation level n.

Example 4. A map function removes `e` if it follows `r`. Thus, the function maps the input word `requirements` to `rquirments`.

3.4 A General Character-removing Map Function

In this section, we develop a general representation of the map functions proposed so far, in Sections 3.2 and 3.3. All map functions have in common that they remove characters from a string based on a condition. Our generalized map function has two parameters:

Observe The length of the character chain to observe, i.e., the characters we use to determine the entropy. We refer to this as o. Each map function requires a character chain of at least one character, i.e., $o > 0$.

Remove The length of the character chain to remove. We refer to this as r with $0 < r \leq o$. For $r < o$, we always remove the characters from the right of the chain observed. In more detail, when observing a character chain c_0c_1, we determine the conditional occurrence probability of character c_1 using $P(c_1|c_0)$. Therefore, we remove character c_1, i.e., the rightmost character in the chain observed.

Our generalized map function allows to simulate all character-removing map functions as follows. We parameterize our generalized map function to observe a single character and, if necessary, remove a single character, i.e., $o = r = 1$. To simulate a character-chain-removing function, we observe and if necessary remove the same number of characters, i.e., $o > 1$ and $r = o$. We simulate a character-removing map function that respects one or more previous characters by observing more characters than we potentially remove, i.e., $o > r$. To remove a single character depending on the two previous characters, we use $o = 3$ and $r = 1$.

Example 5. To remove character `t` if it occurs after the characters `m`, `e`, and `n` (in this order), we specify a map function that observes character chains of length four ($o = 4$) and removes individual characters ($r = 1$). So this function takes the previous $4 - 1 = 3$ characters into account to decide whether to remove character `t` or not. Technically, the map function searches for the four-digit character chain `ment` and replaces it with the chain `men`.

We refer to a map function that observes and removes single characters as `o1r1`, to ones that additionally observe one previous character and remove one character as `o2r1`, and so on. The map function in Example 5 is `o4r1`.

3.5 Cases of Approximation Errors

Character-removing map functions may lead to two sources of approximation error, by (1) conflating tree branches and (2) by the character reduction itself.

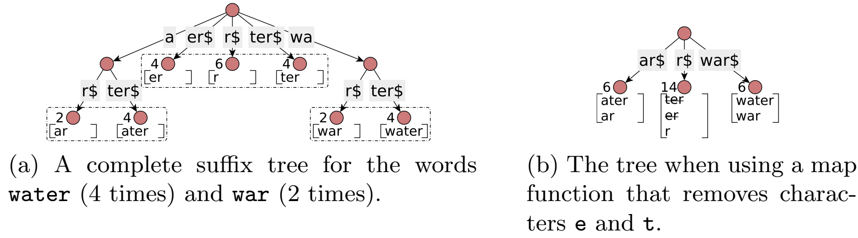
Case 1: Branch Conflation. In most cases, when removing characters, words still map to different strings and, thus, are represented differently in the tree. But in some cases, different words map to the same string and these words correspond to the same node. For example, when removing the characters `e` and `t`, the map function maps `water` and `war` to the same string `war`. The occurrences of the two words are counted in the same node, the one of `war`. Thus, the tree node of `war` stores the sum of the numbers of occurrences of `war` and `water`. So TST estimates the same cardinality for both words.

Case 2: Character Reduction. A character-removing map function shortens most words. But since it removes the most frequent characters/character chains first, it tends to keep characters with high information value. However, with a very high approximation degree, a word may be mapped to the empty string. Our estimation in this case is the count of the root node, i.e., the total number of strings in the tree.

4 Our Approach for Error Correction

Since we investigate the causes of estimation errors, we now develop an approach to correct them. Our approach is to count the number of different input strings that conflate to a tree node. Put differently, we count the number of different preimages of a node. To estimate the string cardinality, we use the multiplicative inverse of the number of different input strings as a correction factor. For example, think of a map function that maps two words to one node., i.e., the node counts the cardinality of both strings. TST estimates half of the node's count for both words.

Example 6. Imagine a suffix tree containing the words `water` and `war`. `water` occurs 4 times and `war` occurs 2 times in the database. Figure 2a shows the full suffix tree. We now build a thin suffix tree by removing characters `e` and `t`. Both words map to the string `war`. Its node holds a count of $4 + 2 = 6$, and the tree will answer both queries, i.e., for `water` and for `war` with 6. Figure 2b shows the corresponding tree. When we query the cardinality of the two words, the relative error is $\frac{6}{4} = 1.5$ for `water` and $\frac{6}{2} = 3$ for `war`. Since we know that this node is reached by 2 different input words, we can answer a query with a cardinality of $6 \cdot \frac{1}{2} = 3$. Using the correction factor $\frac{1}{2}$ reduces the relative error to $\frac{3}{4} = 0.75$ for `water` and $\frac{3}{2} = 1.5$ for `war`.



(a) A complete suffix tree for the words `water` (4 times) and `war` (2 times).

(b) The tree when using a map function that removes characters `e` and `t`.

Fig. 2. Each node of the thin suffix tree includes a suffix count and, additionally to calculate the correction factor, a Bloom filter.

4.1 Counting the Branch Conflations

To compute the correction factor, we need the number of different input words that map to a node. To prevent the tree from double counting input words, each node has to record words already counted. A first idea to do this may be to use a hash table. However, this would store the full strings in the tree nodes and increase the memory usage by much. The directed acyclic word graph (DAWG) [6] seems to be an alternative. It is a deterministic acyclic finite state automaton that represents a set of strings. However, even a DAWG becomes unreasonably large, to be stored in every node [7]. There also are approximate methods to store a set of strings. In the end, we choose the Bloom filter [5] for the following reasons: Firstly, it needs significantly less memory than exact alternatives. Its memory usage is independent of the number as well as of the length of the strings. Secondly, the only errors are false positives. In our case, this means that we may miss to count a preimage. This can lead to a slightly higher correcting factor, e.g., $\frac{1}{3}$ instead of $\frac{1}{4}$ or $\frac{1}{38}$ instead of $\frac{1}{40}$. As a result, the approximate correction factor is always larger than or equal to the true factor. This means that our correction factor only affects the estimation in one direction, i.e., it only corrects an overestimated count.

To sum up, our approach to bring down estimation errors has the following features: For ambiguous suffixes, i.e., ones that collide, it yields a correction factor in the range $(0, 1)$. This improves the estimation compared to no correction. For unambiguous suffixes, i.e., no collision, the correction factor is exactly 1. This means that error correction does not falsify the estimate.

4.2 Counting Fewer Input Strings

TST stores image strings, while our error correction relies on preimage strings. Since the preimage and the image often have different numbers of suffixes (they differ by the number of removed characters), our error correction may count too many different suffixes mapped to a node. For example, take the preimage `water`. A map function that removes characters `a` and `t` returns the image `war`. The preimage `water` consists of 5 suffixes, while the image `war` consists of 3. This renders the correction factor too small and may result in an underestimation. Example 7 illustrates how to avoid counting too many preimage strings.

Example 7. Think of the input word `water` and its suffixes `ater`, `ter`, `er`, and `r`. We use a map function that removes the characters `e` and `t`. This will create a thin suffix tree with nodes that represent `war`, `ar`, and `r`. See Figure 2b. The preimage suffixes `ter`, `er`, and `r` are mapped to the same node. Since `ter` and `er` are mapped to the same string as their next shorter suffix, i.e., `er` and `r` respectively, we do not count these suffixes for the correction factor. The only suffix we count in Node `r` is suffix `r`. All this yields a thin tree with three nodes and a correction factor of $\frac{1}{1}$ in every node.

We count too many preimages iff a preimage suffix maps to the same image as its next shorter suffix. This is the case for every preimage suffix that starts with a character that is removed by the map function. We call the set of characters a map function removes *trim characters*. This lets us discern between two cases: First, the map function removes the first character of the suffix (and maybe others). Second, the map function keeps the first character of the suffix and removes none, exactly one or several characters within it. To distinguish between the two cases, we check whether the first character of the preimage suffix is a trim character. This differentiation also applies to complex map functions that reduce multiple characters from the beginning of the suffix. To solve the issue of counting too many different preimages, our error correction only counts preimage strings which do not start with a trim character.

5 Insert and Query

After describing the details of TST, we now turn to the implementation. We first cover the *map function* and then describe our realization of the functions *insert* and *query*.

5.1 Map Function

Algorithm 1 shows an implementation of our generalized character-removing map function. The function is parametrized by a dictionary that defines which characters to observe and which ones to remove. For example, it contains the three most frequent chains of a length of two characters. For English words, these are `th`, `he`, and `in`. To this end, it removes character `h` if it occurs after character `t`, `e` if it occurs after `h`, and `n` if it occurs after `i`. The map function manipulates only strings that include one of these chains.

5.2 Insert Function

Algorithm 2 inserts a new word into the TST. It consists of the following steps: (1) map the string from the input alphabet to the tree alphabet (Line 3), (2) insert all its suffixes into the tree (Line 4), and (3) add the associated suffix in input alphabet to the Bloom filter of the respective end node of the tree (Line 9).

Algorithm 1: Function to map words from an input alphabet to a string in tree alphabet.

```
1  Function map(inputstring):
       Data: String in input alphabet
       Result: String in tree alphabet
2      dict ⟵ getGlobalDictionary()
       /* E.g., {'th' : 't', 'he' : 'h', 'in' : 'i'}                    */
3      imagestring ⟵ inputstring
4      foreach (k, v) ∈ dict do
5          imagestring ⟵ imagestring.replace (k, v)
6      return imagestring
```

Algorithm 2: Function to insert a string.

```
1  Function insert(inputstring):
       Data: String to add to the tree
2      foreach suffix of inputstring do
3          imagestring ⟵ map(suffix)
4          insertionpath ⟵ Go down the tree path according to imagestring
5          foreach node n on insertionpath do
6              n.count ⟵ n.count + 1
7              if suffix[0] equals imagestring[0] then
8                  if not suffix in n.bloomFilter then
9                      n.bloomFilter.add(suffix)
10                     n.nbDiffSuffixes ⟵ n.nbDiffSuffixes + 1
```

5.3 Query Function

Algorithm 3 is the implementation of how to query the TST. It contains the following three steps: (1) map the given string from input alphabet to tree alphabet (Line 2), (2) search the tree node for this string using the same map function as for function *insert* (Line 3), and (3) estimate the frequency of the string by taking the number of node visits during insertion divided by the number of different suffixes corresponding this node (Line 6).

6 Experimental Evaluation

In this section, we evaluate our thin suffix tree approach. We (1) define the objectives of our evaluation, (2) describe the experimental setup, and (3) present and discuss the results.

6.1 Objectives

The important points of our experiments are as follows.

Algorithm 3: Function to query the cardinality of a string pattern.

```
1 Function query(inputstring):
    Data: String or regular expression
    Result: Cardinality estimation for inputstring
2   imagestring ⟵ map(inputstring)
3   node n ⟵ Go down the tree path according to imagestring
4   if n not exists then
5       return 0
6   return n.count/ n.nbDiffSuffixes
```

Memory Usage We examine the impact of our map functions on the size of the suffix tree.

Map Function We study the effects of our map functions on the estimation accuracy and analyze the source of estimation errors.

Accuracy We investigate the estimation accuracy as function of the tree size.

Query Run Time We evaluate the query run times, i.e., the average and the distribution.

We rely on two performance indicators: *memory usage*, i.e., the total tree size, and the *accuracy* of the estimations. To quantify the accuracy, we use the q-error, a maximum multiplicative error commonly used to properly measure the cardinality estimation accuracy [11, 31]. The q-error is the preferred error metric for cardinality estimation, since it is directly connected to costs and optimality of query plans [32]. Given the true cardinality $\hat{f}$ and the estimated cardinality f, the q-error is defined as follows.

$$\text{q-error} := \max\left(\frac{f}{\hat{f}}, \frac{\hat{f}}{f}\right) \tag{7}$$

6.2 Setup

Our intent is to benchmark the approaches in a real-world scenario. In addition, we inspect and evaluate the impact of different character-removing map functions on the tree. We now describe the database and the queries used in the experiments.

Database. For pattern search on word chains, we use the 5-grams from the English part of the Google Books Ngram corpus.[1] We filter all words that contain a special character, like a digit.[2] At the end, we randomly sample the data set to contain 1 million 5-grams.

[1] The Google Books Ngram corpus is available at http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

[2] We use Java's definition of special characters. See function `isLetter()` of class `Java.lang.Character` for the full definition.

Queries. Users may be interested in querying the number of 5-grams that start with a specific word or a specific word chain. Others may want to query for the number of different words that are used together with a specific word or word chain. The answer to both types of question is the cardinality of a search pattern. For our evaluation, we create 1000 queries requesting the cardinality of the 1000 most common nouns in the English language. For example, we query the number of 5-grams containing words like `way`, `people`, or `information`.

Parametrization. Each TST uses a single character-removing map function. The indicators o and r specify the character-removing map function, i.e., how a map function works. Each function has a character frequency list that stores all characters in descending order of their frequency. Hence, we use the list provided by Peter Norvig in *English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU*[3]. To specify the approximation level of the suffix tree, we parametrize the chosen character-removing map function with the number of characters x that are removed by it. This means that the function takes the first x characters from the character frequency list to specify the set of characters removed by the map function. The competitor has the level of approximation as parameter. The parameter specifies the maximal length of the suffixes to store in the tree. Depending on the string lengths of the database, reasonable parameter values lie between 50 and 1 [39].

Technical Details. Our implementation makes use of SeqAn[4], a fast and robust C++ library. It includes an efficient implementation of the suffix tree and of the suffix array together with state-of-the-art optimizations. We run our experiments on a AMD EPYC 7551 32-Core Processor with 125 GB of RAM. The machine's operating system is an Ubuntu 18.04.4 LTS on a Linux 4.15.0-99-generic kernel. We use the C++ compiler and linker from the GNU Compiler Collection in version 5.4.0.

6.3 Experiments

We now present and discuss the results of our experiments.

Experiment 1: Memory Usage In Experiment 1, we investigate how a character-removing map functions affects the memory consumption of a TST. We also look at the memory needs of a pruned suffix tree (PST) and compare the two. In our evaluation, we inspect chains of lengths up to 3 ($o = \{1, 2, 3\}$) and in each case remove 1 to o characters. Figure 3 shows the memory usage of the TST for various map functions and approximation levels. The figure contains four plots. The first three (from the left) show the memory usage for map functions that observe character chains of length 1, 2, or 3. The right plot shows the memory usage of the pruned suffix tree contingent on the maximal length of

[3] The article and the list are available at https://norvig.com/mayzner.html
[4] The SeqAn library is available at https://www.seqan.de.

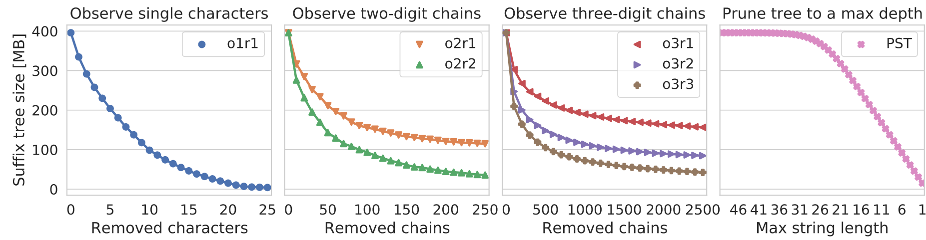


Fig. 3. TST's memory usage for various map functions and approximation levels.

the suffixes. Note that there are different scales on the x-axis. The database we use in this evaluation includes 179 different characters, 3,889 different character chains of length two, and 44,198 different chains of length three.

The Effect of Character-Removing Map Functions. For a deeper insight into our pruning approach, we now study the effect of character-removing map functions on the memory consumption of a suffix tree in more detail. As discussed in Section 3.5, there are two effects that reduce memory consumption: *branch conflation* and *character reduction*. See Figure 3. All map functions yield a similar curve: They decrease exponentially. Hence, removing the five most frequent characters halves the memory usage of a TST with map function `o1r1`.

The frequency of characters and character chains in natural language follows Zipf's law, i.e., the Zeta distribution [39]. Zipf's law describes a relation between the frequency of a character and its rank. According to the distribution, the first most frequent character nearly occurs twice as often as the second one and so on. All character-removing map functions in Figure 3 show a similar behavior: Each approximation level saves nearly half of the memory as the approximation level before. For example, map function `o1r1` saves nearly 65MB from approximation level 0 to 1 and nearly 40MB from approximation level 1 to 2. This shows the expected behavior, i.e., character reduction has more impact on the memory usage of a TST than branch conflation.

The Effect of Horizontal Pruning. The right plot of Figure 3 shows the memory usage of a pruned suffix tree. The lengths of words in natural language are Poisson distributed [39]. In our database, the strings have an average length of 25.9 characters with a standard deviation of 4.7 characters. Since the pruning only affects strings longer than the maximal length, the memory usage of the pruned suffix tree follows the cumulative distribution function of a Poisson distribution.

Summary. Experiment 1 reveals two points. First, the tree size of TST and of the pruned suffix are markedly different for the various approximation levels. Second, the memory reduction of a TST is independent of the length of the strings in the database. Its memory reduction depends on the usage frequency

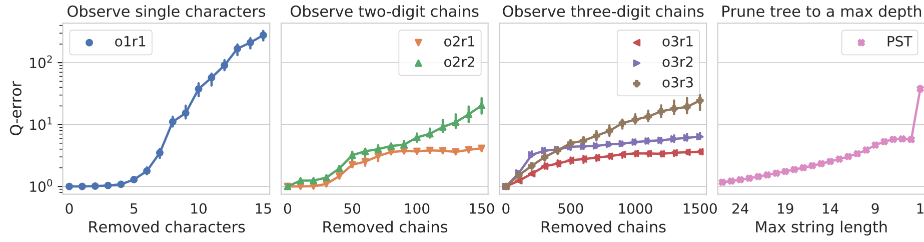


Fig. 4. TST's q-error for various map functions and approximation levels.

of characters in natural language. Third, the tree sizes for the different character-removing map functions tend to be similar, except for one detail: The shorter the chain of observed characters, i.e., the smaller o, the more linear the reduction of the tree size over the approximation levels.

Experiment 2: Map Functions In Experiment 2, we investigate the impact of map functions on estimation accuracy. We take the map functions from Experiment 1 and measure the q-error at several approximation levels. See Figure 4. The right plot is the q-error with the pruned suffix tree. The points are the median q-error of 1000 queries. The error bars show the 95% confidence interval of the estimation. Note that Figure 4 shows the estimation performance as a function of the approximation level of the respective map function. So one cannot compare the absolute performance of different map functions, but study their behavior. The plots show the following. First, the map functions behave differently. At low approximation levels, the map function `o1r1` has a very low q-error. The q-error for this function begins to increase slower than for map functions considering three-digit character chains. At high approximation levels, the q-error of map function `o1r1` is significantly higher than for all the other map functions. The other map functions, the ones that consider three-digit character chains in particular, only show a small increase of the q-error for higher approximation levels. Second, the sizes of the confidence interval differ. With map functions that remove characters independently from previous characters, i.e., `o1r1`, `o2r2`, and `o3r3`, the confidence interval becomes larger with a larger approximation level. For map functions that remove characters depending of previous characters, i.e., `o2r1`, `o3r1`, or `o3r2`, the confidence interval is smaller. This means that, for lower approximation levels and map functions that remove characters depending on previous characters in particular, our experiments yield reliable results. We expect results to be the same on other data. Third, the accuracy of the pruned suffix tree increases nearly linear with increasing approximation levels until it sharply increases for very short maximal strings lengths. This means that every character that is removed from the back of the suffix contributes a similar extent of error to an estimation.

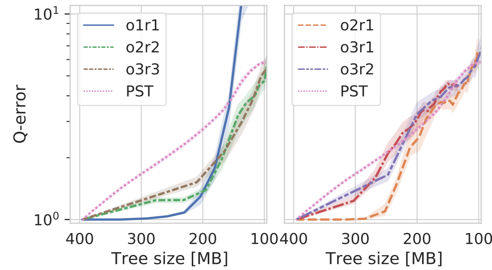


Fig. 5. The estimation accuracy as function of the tree size.

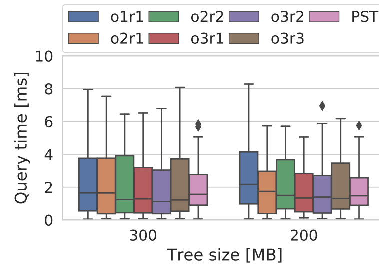


Fig. 6. The query run time of the TST.

Summary. None of our map functions dominates all the other ones. It seems to work best to remove single characters dependent on either 0, 1, or 2 previous characters, i.e., map functions `o1r1`, `o2r1`, or `o3r1`. For low approximation levels, say up to a reduction of 50% of the tree size, map function `o1r1` performs well. For high approximation levels, say starting from a reduction of 50% of the tree size, one should use a map function that removes characters dependent on previous ones, i.e., map function `o2r1` or `o3r1`.

The first five approximation levels of map function `o1r1` are of particular interest, as they show good performance in Experiments 1 and 2. In the first approximation levels, this map function yields a very low q-error, see Figure 4, while the tree size goes down very rapidly, see Figure 3. In Experiments 1 and 2, we inspect (1) the tree size depending on the approximation and (2) the q-error depending on the approximation level. In many applications, one is interested in the q-error depending on the tree size rather than on the approximation level. In our next experiment, we compare the q-error of our map functions for the same tree sizes.

Experiment 3: Accuracy In Experiment 3, we compare the map functions from Experiment 1 against each other and against existing horizontal pruning. Figure 5 shows the q-error as function of the tree size. For the sake of clarity, there are two plots for this experiment: The plot on the left side shows the map functions that remove characters independently from previous characters. The plot on the right side is for the remaining map functions.

Vertical Pruning vs. Horizontal Pruning. Figure 5 shows the following: TST produces a significantly lower q-error than the pruned suffix tree for all map functions used and for tree sizes larger than 60% of the one of the full tree. For smaller sizes, the accuracy of most map functions does not become much worse than the one of the pruned suffix tree. The only exception is `o1r1`. At a tree size of 50%, the q-error of `o1r1` starts to increase exponentially with decreasing tree size.

Summary. As Experiments 1 and 2 already indicate, map function `o1r1` achieves a very low q-error to tree size ratio, for tree size reductions of up to 60%. For this map function, TST yields a significantly lower q-error than the pruned suffix tree for comparable tree sizes. The intuition behind this result is that our vertical pruning respects the redundancy of natural language. Due to this redundancy, map function `o1r1` keeps most input words unique. This results in almost no errors for reductions of the tree size that are less than 50%. TST also shows a higher degree of confidence, i.e., a smaller 95% confidence interval, than the pruned suffix tree for reductions that are less than 40%.

Experiment 4: Query Run Time In Experiment 4, we compare the query run time of the TST using different map functions with the one of a pruned suffix tree. We consider sample tree sizes of 300 and 200MB. Figure 6 shows the average and distribution of the run time for all queries. There is no significant difference in the run time. TST potentially needs a slightly higher run time than a pruned suffix tree. This is because TST is potentially deeper than a pruned suffix tree and additionally executes a map function. To conclude, the additional work of TST is of little importance for the query run time compared to a pruned suffix tree.

7 Conclusions

Cardinality estimation for string attributes is challenging, for long strings in particular. Suffix trees allow fast implementations of many important string operations, including this estimation. But since they tend to use much memory, they usually are pruned down to a certain size. In this work, we propose a novel pruning technique for suffix trees for long strings. Existing pruning methods mostly are horizontal, i.e., prune the tree to a maximum depth. Here we propose what we call *vertical pruning*. It reduces the number of branches by merging them. We define map functions that remove characters from the strings based on the entropy or conditional entropy of characters in natural language. Our experiments show that our thin suffix tree approach does result in almost no error for tree size reductions of up to 50% and a lower error than horizontal pruning for reductions of up to 60%.

References

1. Adams, E., Meltzer, A.: Trigrams as index element in full text retrieval: Observations and experimental results. In: CSC. pp. 433–439. ACM (1993). https://doi.org/10.1145/170791.170891
2. Andersson, A., Nilsson, S.: Improved behaviour of tries by adaptive branching. Information Processing Letters pp. 295–300 (1993). https://doi.org/10.1016/0020-0190(93)90068-k
3. Arz, J., Fischer, J.: LZ-compressed string dictionaries. In: DCC. IEEE (2014). https://doi.org/10.1109/dcc.2014.36

4. Bille, P., Fernstrøm, F., Gørtz, I.: Tight bounds for top tree compression. In: String Processing and Information Retrieval, pp. 97–102. Springer (2017). https://doi.org/10.1007/978-3-319-67428-5_9
5. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM pp. 422–426 (1970). https://doi.org/10.1145/362686.362692
6. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M., Seiferas, J.: The smallest automation recognizing the subwords of a text. Theoretical Computer Science pp. 31–55 (1985). https://doi.org/10.1016/0304-3975(85)90157-4
7. Blumer, A., Ehrenfeucht, A., Haussler, D.: Average sizes of suffix trees and DAWGs. Discrete Applied Mathematics pp. 37–45 (1989). https://doi.org/10.1016/0166-218x(92)90270-k
8. Brown, P., Della, V., Mercer, R., Pietra, S., Lai, J.: An estimate of an upper bound for the entropy of english. Comput. Linguist. pp. 31–40 (1992)
9. Chaudhuri, S., Ganti, V., Gravano, L.: Selectivity estimation for string predicates: Overcoming the underestimation problem. In: ICDE. IEEE (2004). https://doi.org/10.1109/icde.2004.1319999
10. Claude, F., Navarro, G., Peltola, H., Salmela, L., Tarhio, J.: String matching with alphabet sampling. Journal of Discrete Algorithms pp. 37–50 (2012). https://doi.org/10.1016/j.jda.2010.09.004
11. Cormode, G., Garofalakis, M., Haas, P., Jermaine, C.: Synopses for massive data: Samples, histograms, wavelets, sketches. Foundations and Trends in Databases pp. 1–294 (2011). https://doi.org/10.1561/1900000004
12. Dorohonceanu, B., Nevill-Manning, C.: Accelerating protein classification using suffix trees. ISMB pp. 128–133 (2000)
13. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms p. 20 (2007). https://doi.org/10.1145/1240233.1240243
14. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Transactions on Algorithms pp. 1–21 (2010). https://doi.org/10.1145/1868237.1868248
15. Gog, S., Moffat, A., Culpepper, S., Turpin, A., Wirth, A.: Large-scale pattern search using reduced-space on-disk suffix arrays. TKDE pp. 1918–1931 (2014). https://doi.org/10.1109/tkde.2013.129
16. Grabowski, S., Raniszewski, M.: Sampling the suffix array with minimizers. In: String Processing and Information Retrieval, pp. 287–298. Springer (2015). https://doi.org/10.1007/978-3-319-23826-5_28
17. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. Journal of Experimental Algorithmics pp. 11–120 (2015). https://doi.org/10.1145/2656332
18. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing pp. 378–407 (2005). https://doi.org/10.1137/s0097539702402354
19. Hu, T., Tucker, A.: Optimal computer search trees and variable-length alphabetical codes. SIAM Journal on Applied Mathematics pp. 514–532 (1971). https://doi.org/10.1137/0121057
20. Huffman, D.: A method for the construction of minimum-redundancy codes. IRE pp. 1098–1101 (1952). https://doi.org/10.1109/jrproc.1952.273898
21. Kanda, S., Morita, K., Fuketa, M.: Practical implementation of space-efficient dynamic keyword dictionaries. In: String Processing and Information Retrieval, pp. 221–233. Springer (2017). https://doi.org/10.1007/978-3-319-67428-5_19

22. Kirschenhofer, P., Prodinger, H.: Some further results on digital search trees. In: Automata, Languages and Programming, pp. 177–185. Springer (1986). https://doi.org/10.1007/3-540-16761-7_67
23. Krishnan, P., Vitter, J., Iyer, B.: Estimating alphanumeric selectivity in the presence of wildcards. ACM SIGMOD Record pp. 282–293 (1996). https://doi.org/10.1145/235968.233341
24. Kroeger, P.: Analyzing Grammar: An Introduction. Cambridge University Press (2015)
25. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Lecture Notes in Computer Science, pp. 219–230. Springer (1996). https://doi.org/10.1007/3-540-61332-3_155
26. Larsson, N., Moffat, A.: Off-line dictionary-based compression. IEEE pp. 1722–1732 (2000). https://doi.org/10.1109/5.892708
27. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? VLDB Endowment pp. 204–215 (2015). https://doi.org/10.14778/2850583.2850594
28. Li, D., Zhang, Q., Liang, X., Guan, J., Xu, Y.: Selectivity estimation for string predicates based on modified pruned count-suffix tree. CJE pp. 76–82 (2015). https://doi.org/10.1049/cje.2015.01.013
29. Manning, C., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press (1999)
30. Miner, G., Elder, J., Fast, A., Hill, T., Nisbet, R., Delen, D.: Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications. Academic (2012)
31. Moerkotte, G., DeHaan, D., May, N., Nica, A., Boehm, A.: Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In: SIGMOD. ACM (2014). https://doi.org/10.1145/2588555.2595629
32. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. VLDB Endowment pp. 982–993 (2009). https://doi.org/10.14778/1687627.1687738
33. Moradi, H., Grzymala-Busse, J., Roberts, J.: Entropy of english text: Experiments with humans and a machine learning system based on rough sets. Information Sciences pp. 31–47 (1998). https://doi.org/10.1016/s0020-0255(97)00074-1
34. Müller, M., Moerkotte, G., Kolb, O.: Improved selectivity estimation by combining knowledge from sampling and synopses. VLDB Endowment pp. 1016–1028 (2018). https://doi.org/10.14778/3213880.3213882
35. Nilsson, S., Tikkanen, M.: An experimental study of compression methods for dynamic tries. Algorithmica pp. 19–33 (2002). https://doi.org/10.1007/s00453-001-0102-y
36. Poyias, A., Raman, R.: Improved practical compact dynamic tries. In: String Processing and Information Retrieval, pp. 324–336. Springer (2015). https://doi.org/10.1007/978-3-319-23826-5_31
37. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems pp. 589–607 (2007). https://doi.org/10.1007/s00224-006-1198-x
38. Sautter, G., Abba, C., Böhm, K.: Improved count suffix trees for natural language data. In: IDEAS. ACM (2008). https://doi.org/10.1145/1451940.1451972
39. Sigurd, B., Eeg-Olofsson, M., van Weijer, J.: Word length, sentence length and frequency - zipf revisited. Studia Linguistica pp. 37–52 (2004). https://doi.org/10.1111/j.0039-3193.2004.00109.x
40. Sun, J., Li, G.: An end-to-end learning-based cost estimator (2019)

41. Vitale, L., Martín, Á., Seroussi, G.: Space-efficient representation of truncated suffix trees, with applications to markov order estimation. Theoretical Computer Science pp. 34–45 (2015). https://doi.org/10.1016/j.tcs.2015.06.013
42. Welch, T.: A technique for high-performance data compression. Computer pp. 8–19 (1984). https://doi.org/10.1109/mc.1984.1659158
43. Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., Naughton, J.: Predicting query execution time: Are optimizer cost models really unusable? In: ICDE. IEEE (2013). https://doi.org/10.1109/icde.2013.6544899
44. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory pp. 337–343 (1977). https://doi.org/10.1109/tit.1977.1055714