Karlsruher Institut für Technologie

Bachelor Thesis

Implementation and Evaluation of an External Memory String B-Tree

Fellipe Bernardes Lima

Published: 29/12/2014

Supervisor: Prof. Dr. Peter Sanders
Dipl. Inform. Timo Bingmann

Institute of Theoretical Computer Science, Algorithmics II
Department of Computer Science
Karlsruhe Institute of Technolgy

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Zusammenfassung

Aufbereitung großer Texten, um Substring-Abfragen zu beantworten ist nicht trivial, wann immer realistische Modelle berücksichtigt werden. Wir bahandeln dieses Problem, indem wir eine effiziente Implementierung vom String B-Tree zur Auflösung vom Problem der Substring-Suche bei den dynamischen Szenarien anbieten. Wir erreichen optimalen Speicherverbrauch für die Patricia Tries durch eine Multiarray-Kodierung und unser Ansatz für den String B-Tree nutzt weniger Speicherplatz als alle andere Volltext-Indizes für externen Speicher vorhanden. Die Darstellung ist strukturell sehr gut geeignet für Entropiekompression und der String B-Tree wird neuartig effizient aufgebaut, nähmlich mit effizienter Laufzeit bezüglich CPU-Verarbeitung und I/O-Operationen. Für dieses Konstruktionsverfahren werden das Suffix Array and das LCP Array eingesetzt, die durch andere effizienten Algorithmen berechnet werden können.

Abstract

Preprocessing texts of huge size to answer substring queries is not trivial whenever considering realistic models. We approach this problem by offering an efficient implementation of the String B-Tree data structure, which aims to solve the substring search problem under the dynamic operations. We achieve optimal space usage for the Patricia Tries by representing them via multiarray encoding and our approach for the String B-Tree uses less space than any other external memory full-text index available. The tree representation is well suited for entropy compression and the String B-Tree is efficiently constructed regarding CPU processing and I/O operations. During that construction process we reuse the SA and the LCP arrays, which can be generated by other efficient algorithms.

Acknowledgement

To Peter Sanders and Timo Bingmann for giving me this special opportunity and strongly contributing to my growth in computer science and personal life.

To Melina Metzig and Ioana Gheţa, who definitively guided my studies in Karlsruhe. Thanks a lot for the patience of clearing every question in the beginning.

To Flavio Cardeal and Giani Silva, who gave me support to come to Germany for studying. Unfortunatelly, I decided to follow the path of theoretical computer science, but still I miss the work we did together and am forever thankful for every help.

To my parents for giving me education and assistence. I am really thankful for everything they have done for me along this time.

To all my colleagues from the software industry, who cooperated so that I could get a strong background about software development. It definitvely helped me a lot indirectly in this thesis and I would be happy if this solution can be useful in their fields.

To all research assitants who offered me student jobs at the university. I am sure that without them I could not learn as much as I did in Karlsruhe.

To the whole Faculty of Informatics for accepting me as an undergraduate student and giving me the chance to study in one of the best universities of the world.

To Paolo Ferragina for providing us this great open theoretical foundation which is a source of interesting questions.

Contents

1 Introduction **8**
- 1.1 Contribution of the Thesis . . . 10
- 1.2 Structure of the Thesis . . . 10

2 Fundamentals **11**
- 2.1 Problem Definition . . . 11
- 2.2 Notations . . . 12
- 2.3 Models . . . 12
- 2.4 B^+-Trees . . . 14
- 2.5 Basic Text Indexes . . . 16
  - 2.5.1 Tries . . . 16
  - 2.5.2 Suffix Arrays . . . 19
  - 2.5.3 Suffix Trees . . . 19
- 2.6 External Memory Text Indexes . . . 21
  - 2.6.1 Hierarchies of Indexes using Suffix Arrays . . . 21
  - 2.6.2 Compact Suffix Trees . . . 21
  - 2.6.3 String B-Tree . . . 22
- 2.7 More Related Work . . . 24

3 String B-Tree **25**
- 3.1 Definitions . . . 25
- 3.2 $\texttt{Search}_{SB}$ Operation . . . 25
- 3.3 $\texttt{Insert}_{SB}$ Operation . . . 26
- 3.4 Extended PTs . . . 28
  - 3.4.1 Serialization Methods . . . 28
  - 3.4.2 $\texttt{GetPosition}_{PT}$ Operation . . . 33
  - 3.4.3 $\texttt{Insert}_{PT}$ Operation . . . 35
  - 3.4.4 $\texttt{Split}_{PT}$ Operation . . . 42
- 3.5 Bulk Construction . . . 43
  - 3.5.1 Patricia Trie . . . 43
  - 3.5.2 String B-Tree . . . 47
- 3.6 Limitations . . . 51

4 Implementation Details **53**
- 4.1 String B-Tree . . . 53
- 4.2 Patricia Trie . . . 53

5 Experimental Results **54**
- 5.1 Experimental Setup . . . 54
- 5.2 Input Data Sets . . . 54
- 5.3 Discussion . . . 54
  - 5.3.1 Patricia Trie . . . 54
  - 5.3.2 String B-Tree . . . 58

6 Conclusion **61**
- 6.1 Future Work . . . 61

List of Figures

1 B-Tree example . . . 15
2 Trie example . . . 17
3 Compact Trie example . . . 17
4 Patricia Trie example . . . 18
5 Suffix Array example . . . 19
6 Suffix Tree example . . . 20
7 Patricia Trie example for serialization . . . 30
8 Byte-based serialization representing the trie of Figure 7 . . . 30
9 Multiarray serialization representing the trie of Figure 7 . . . 31
10 Higher jump pointers for multiarray serialization . . . 32
11 Lower jump pointers for multiarray serialization . . . 32
12 Trie transformation: $\texttt{PutBranchForInternal}_{\texttt{PT}}$ . . . 38
13 Trie transformation: $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ . . . 38
14 Trie transformation: $\texttt{PutLeafForInternal}_{\texttt{PT}}$ . . . 38
15 Example of trie transformation: $\texttt{PutBranchForInternal}_{\texttt{PT}}$ operation . . . 39
16 Before $\texttt{PutBranchForInternal}_{\texttt{PT}}$ (MA) . . . 39
17 After $\texttt{PutBranchForInternal}_{\texttt{PT}}$ (MA) . . . 39
18 Before $\texttt{PutBranchForInternal}_{\texttt{PT}}$ (BB) . . . 39
19 After $\texttt{PutBranchForInternal}_{\texttt{PT}}$ (BB) . . . 39
20 Example of trie transformation: $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ operation . . . 40
21 Before $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ (MA) . . . 40
22 After $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ (MA) . . . 40
23 Before $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ (BB) . . . 40
24 After $\texttt{PutBranchForLeaf}_{\texttt{PT}}$ (BB) . . . 40
25 Example of trie transformation: $\texttt{PutLeafForInternal}_{\texttt{PT}}$ operation . . . 41
26 Before $\texttt{PutLeafForInternal}_{\texttt{PT}}$ (MA) . . . 41
27 After $\texttt{PutLeafForInternal}_{\texttt{PT}}$ (MA) . . . 41
28 Before $\texttt{PutLeafForInternal}_{\texttt{PT}}$ (BB) . . . 41
29 After $\texttt{PutLeafForInternal}_{\texttt{PT}}$ (BB) . . . 41
30 Patricia Trie example for bulk construction . . . 44
31 Correspondance between the SA and LCP arrays, and the Patricia Trie . . . 44
32 Analysis of the $\texttt{Insert}_{\texttt{PT}}$ operation . . . 55
33 Analysis of the $\texttt{Search}_{\texttt{PT}}$ operation . . . 56
34 Analysis of the $\texttt{BulkConstruction}_{\texttt{PT}}$ operation . . . 56
35 Analysis of the Patricia Trie operations . . . 57
36 Analysis of the Patricia Trie's space consumption . . . 57
37 Analysis of the String B-Tree regarding the absolute number of I/Os . . . 59
38 Analysis of the String B-Tree regarding the absolute running time . . . 60
39 Analysis of the String B Tree's space consumption . . . 60

List of Tables

1 Worst-case summary table for the basic indexes . . . 21
2 Worst-case summary table for the external memory indexes . . . 23
3 Worst-case summary table for the Patricia Trie representations . . . 33
4 Absolute index growth per insertion transformation . . . 42
5 Worst-case space usage savings of the MA serialization against the BB serialization 42

6 Worst-case summary table for the bulk construction algorithms 51

Algorithmenverzeichnis

1 $\texttt{Search}_{\mathsf{PT}}$ operation 26
2 $\texttt{Insert}_{\mathsf{SB}}$ operation 26
3 $\texttt{InsertUpDown}_{\mathsf{SB}}$ auxiliary function of the $\texttt{Insert}_{\mathsf{SB}}$ operation 27
4 $\texttt{BlindSearch}_{\mathsf{PT}}$ operation 34
5 $\texttt{GetPosition}_{\mathsf{PT}}$ operation 36
6 $\texttt{Insert}_{\mathsf{PT}}$ operation 37
7 $\texttt{GetRanges}_{\mathsf{PT}}$ auxiliary function of the $\texttt{BulkConstruction}_{\mathsf{PT}}$ operation 45
8 $\texttt{BulkConstruction}_{\mathsf{PT}}$ operation 46
9 $\texttt{BulkConstruction}_{\mathsf{SB}}$ operation 49
10 $\texttt{BulkConstruction}_{\mathsf{SB}}$ auxiliary function of the $\texttt{BulkConstruction}_{\mathsf{SB}}$ operation . . 50

1 Introduction

With the expansion of the World Wide Web and the advent of modern mass storage devices, text indexing becomes more feasible and turns out to be one of the most prominent topics in computer science. The theoretical backgrounds are solid and have been worked out since the begin of the 20th century [10]. At the practical side, many solutions exist, but still one needs to improve performance aspects with provable guarantees, in order to fulfill industrial and academic goals. A solution providing fast access to huge masses of text via *queries* requires a good understanding of the foundations of information theory, data management and algorithmics.

An obvious option in searching for those basic queries is to scan the text sequentially and a second one is to build data structures over the text, the indexes, to speed up the search operation. In this sense it worths bulding and maintaining such an index in case the text collection is *large* and *semi-static* [41], because the large text doesn't need to be completely considered during search and the dynamic operations doesn't need to be regarded as the most important case, since the text is not changed frequently. Not to traverse the whole text is obviously much better than traversing it and avoiding update operations usually implies less extra space usage.

The main techniques needed to implement those query operations are based on *indexing* and we focus on *full-text indexing*, that is indexing text sources so that all substrings can be found via queries. One first tries to search queries composed of words and reporting the documents where they are found. The number of occurances of a query in each document and their exact positions might be requested. Algorithms answering those questions consider then *pattern matching* in arbitrary texts [30].

Information retrieval deals with the representation, storage, organizaton, and access to the information items. The representation and organization of the information items aims to provide easy and fast access. Users firstly translate their information need into a *query* which can be processed by a search engine [41]. The result of a query could be for example a list of documents containing a given phrase or the specific locations within a text containing them. Although the mentioned algorithms and data structures for solving text indexing problems don't constraint the input's format, not all the properties and guarantees of those conventional methods can be directly used whenever using realistic models, like the external memory model or the network model [21].

Structuring, storing, evaluating and analyzing large text data sets needs more than basic algorithms because of scalability. Thus, understanding the hardware and the network architecture, designing new algorithms using more realistic models and evaluating running time as well as space consumption are essential ingredients for fulfilling the technical needs of text indexing.

An *inverted file (or inverted index)* is a word-oriented mechanism for indexing text collections in order to speed up the searching task. It consists of two elements: the *vocabulary* and the *occurances*. The set of positions where the words are coming from are listed for each vocabulary word. Here one notices that storing the occurances requires much space, so that the extra space is linear in the number of words of the text. To reduce space requirements, *block addressing* can be used, but still it assumes that the text can be considered as a sequence of words. Other data structures can be introduced that consume much less space and provide more powerful queries, still being simple to understand.

In order to deal with the restrictions of the inverted files regarding the input structure, *suffix trees* and *suffix arrays* can be used, being good examples of powerful and compact data structures. This kind of index allows us to efficiently answer more complex queries and don't restrict the texts to be word-based. Their main drawbacks are the costly construction processes of those data structures.

In essence, a *suffix tree* (ST) is a trie data structure built over all the suffixes of a text. To improve space utilization *Patricia Tries* (PTs) offers a way of compression, so that the data structure turns out to have $\mathcal{O}(n)$ nodes, instead of $\mathcal{O}(n^2)$ of tries [41], where n is the number of stored keys.

Suffix arrays (SAs) provide similar functionality as the STs, but require much less space at cost of more expensive search operations. The contents of a SA are the string pointers resulting of traversing the leaves of the corresponding ST using a depth-first search [49].

Beside those basic data structures, probabilistic data structures account for unexact queries, like *signature files* and its variants. However, the dynamic case fails and we aim to study the case of *exact* pattern matching in this thesis. In what follows, further applications can also be realized upon data structures used for pattern matching. Examples are *string matching allowing errors* and *extended patterns via regular expressions*. Their derivations are relatively straightforward with the conventional data structures, so that the main bottlenecks concern the basic toolbox for text indexing mentioned above.

Traditional text indexes also don't consider word-based data sets. The former are usually no kind of natural language texts structured by words and respecting the rules of a language, but a type of information where no definition about the content structure is given (e.g. the human genome or a list of symbols representing natural phenomena). The latter could be exemplified by a collection of scientific papers or the published articles of a magazine, that are accessible through the Web.

Whenever considering different kinds of written information, that is a data-type independent information source, maximizing efficiency in respect to memory accesses on computer systems is very important. Organizing the data in such a way that it fits the given hardware features turns out to be decisive, meanwhile analyzing worst-case scenarios is also an important aspect for sake of scalability.

Most of the work on full-text indexes has been done on the RAM model, but not always the text fits into the internal memory and must be stored either in the hard disk or transmitted via network. Not only the text size is a determining factor, but also the *full-text index's size*, that is usually 4-20 times larger than the size of the text itself, dependending on implementation and design decisions.

It is well-known that full-text indexes have poor memory locality, because the index structure depends on the text structure and distribution, i.e. the substring repetitions and their locations in the text. Thus, adapting full-text indexes to the external memory model turns out to be mandatory in high performance scenarios.

There are two main issues to be evaluated in full-text indexes: firstly the operations must be I/O efficient, preferably also following the absolute performance guaranteed by conventional internal memory data structures for all input lengths. A second issue is the process of construction of those indexes. For the latter issue, there are few related work published, no robust implementation is available and the related algorithms are unnecessarily complicated.

Beside basic indexes, discussed in the first chapters, a flexible data structure called String B-Tree [34] had been proposed to fill many technical gaps when dealing with the problem of *string matching* in large data sets. Other dissertations approach the problem as well [24, 8] and provide open-access implementations, but still there is a need for more refined results. No tested implementation is open and many important aspects are still not handled, for example the minimization of space consumption and the fast index construction. Experimental results systematically using the diverse combinations of the data structure's parameters are also not provided and hardly can be reproduced. This all makes comparison with other data structures more difficult and doesn't allow for more progress.

1.1 Contribution of the Thesis

As a first step of this work, the String B-Tree [34] is studied considering implementation details, the underlying theoretical aspects and its relationships to other conventional data structures used for full-text indexing. Beside a flexible implementation, some variants of the elementary building blocks (e.g. the internal node representation) are redefined, newly implemented, analyzed and tested.

We describe results in terms of experimental numbers and facts, making sure that analysis is concise and reliable. As a further result we provide a review of the available solutions in both theoretical and practical aspects, giving concrete examples.

In this work, we target the *dynamic* scenario of pattern matching using realistic models, considering the minimization of *space usage* subordinated to fast *search* operations on large input data sets. The index should support updates in a fast and flexible way, without requiring its reconstruction, and the most used operation, the `Search` operation, should be very fast in diverse computation models and not degenerate as the index grows.

1.2 Structure of the Thesis

Section 2 gives a general view about the used notations and existing solutions for the problem of *pattern matching* in realistic models. We review the theoretical and practical aspects of the available basic indexes and check the elementary building blocks of the String B-Tree.

Although the String B-Tree's original definition [34] is the basis for this work, in section 3 we modify some specific details of the algorithms in order to simplify implementation and ease analysis. Moreover, it allows for a slightly better space usage without affecting correctness.

In section 4, we focus on the implementation details of the algorithms and data structures studied in this thesis. We establish how did we get practical performance improvements using a programming language.

The practical correctness and performance of our C++ implementations are evaluated in sections 5, where the speed up of the different variants of our algorithms are compared. In the last section, final remarks are given and an outlook for future works is provided.

2 Fundamentals

At first, we formally define the problems to be solved, present some notations and introduce the models being used. Next, we review the fundamental B-Tree data structure, which is the basis for this work. We then briefly study some of the basic text indexes, which are claimed to efficiently solve small problem instances in main memory. The last part of this chapter reviews the related work, contextualizing the problem and introducing parts of the state-of-the-art.

2.1 Problem Definition

The major problem considered in full-text indexing can be defined as follows in terms of the `Search` operation.

Problem 2.1. ***(Substring Search)*** *[21] Let the text $\mathcal{T}$ be a set of K strings in Σ^* with total length N. A string matching query on the text is: Given a pattern $P \in \Sigma^*$, find all occurances of P in the text $\mathcal{T}$, where Σ is the text alphabet.*

The key ingredient for defining the `Search` operation is that if an occurance of a pattern P starts at position i in a string $S \in \mathcal{T}$, then P is a prefix of the suffix $S[i, |S|]$. In this sense, the task is reduced to perform *prefix search queries* on the set of all suffixes of the text. The dynamic version of the problem additionally requires support for the operations `Insert` and `Delete` of strings into/from $\mathcal{T}$.

Considering the specific problems to be solved in this thesis, the construction of PTs and String B-Trees is an important issue. An *iterative* procedure for the construction of those data structures is based on the successive execution of the `Insert` operation. For String B-Trees, this is not only prohibitive in terms of execution time because of the huge number of I/O operations, but it also generates fragmented tree representations, since heuristics must applied.

A *bulk* procedure for the construction of those data structures can use the `SA` and `LCP` arrays in order to build the index and is usually faster than the iterative construction algorithm. The SA and LCP arrays represent together the ST in a natural and compact way. Next, the problems to be solved are formally defined.

Problem 2.2. ***(Patricia Trie Construction)*** *Let the text $\mathcal{T}$ be a set of K strings in Σ^* with total length N. Given the SA of the text $SA_{\mathcal{T}}$ and the corresponding $LCP_{\mathcal{T}}$ array, construct the corresponding PT using optimal space-consumption, while supporting the operations `Search`$_{\texttt{PT}}$, `Insert`$_{\texttt{PT}}$ and `Delete`$_{\texttt{PT}}$ in $\mathcal{O}(|P| \log |\Sigma|)$ internal time and $\mathcal{O}(1)$ I/Os.*

In the problem definition above, we assume that the SA of the text is the SA of the string resulting of concatenating all strings. The mentioned optimal space-consumption regards the minimal amount of bits used to represent a multiway tree without applying any domain compression technique, that is entropy compression algorithms are not considered through the whole text and this issue is postponed to the future work.

The problem above defines the tasks of constructing a single PT using the `SA` and `LCP` arrays or subparts of them. However, we are also interested in generating the complete String B-Tree structure, so that a third problem must be defined.

Problem 2.3. ***(String B-Tree Construction)*** *Let the text $\mathcal{T}$ be a set of K strings in Σ^* with total length N. Given the $SA_{\mathcal{T}}$ and $LCP_{\mathcal{T}}$ arrays, construct the corresponding String B-Tree due Ferragina [32].*

We aim to solve all problems defined above. Observe that in case we are given an array of sorted strings, rather than the SA, accompanied by the array of LCP values, the algorithms should equally work. Thus, not only the `Search` operations are investigated here, but also the operations of the dynamic case as well as the methods to set up the indexes.

In this work, the Problem 2.2 turns out to be a subproblem of the Problem 2.3 and we the details are discussed in the section 3. Efficient methods for constructing the String B-Tree are not described in other papers and the concurrent solutions are mostly designed for efficient construction of STs.

2.2 Notations

Along this work we follow the following notation in order to unify the work done by many parts and to introduce our algorithms. A big part of our notation stems from the original paper [34], although other notations also influenced the notation of this thesis. All over the thesis, the term internal work regards character comparisons done by the CPU, as an I/O operation regards the transfer of a block of memory words between two memory levels involving random accesses.

We aim to store a set of text strings $\mathcal{S} = \{\delta_1, ..., \delta_K\}$ drawn from a totally ordered alphabet Σ with size $\sigma = |\Sigma|$, where Σ^* denotes all words generated by the alphabet and $\leq_L$ denote the lexicographic total ordering among the strings, introduced by using the characters taken from Σ. The total length of the texts is $N = \sum_{i=1}^{K} |\delta_i|$.

Whenever dealing with tries (or their derivatives), we aim to hold key-value pairs with keys coming from a set $\mathcal{S}$. $parent(u)$ denotes the parent of a given node u and $length(l)$ denotes the length of the string represented by a leaf l. For any string s, we let $s[\mathrm{i}]$ be its i-th character and s_i the suffix starting at the position i of the string. Further we denote s_{ij} the substring starting at the index i and terminating at the index j. Given a node u, $string(u)$ is the string stored at that node u. Whenever using a data structure, P denotes a query and $p = |P|$ its size.

An important operation used along the work is the *rank* operation used to calculate jump offsets on-the-fly. The operation $\texttt{rank}_c(i)$ returns the count of symbols c from the array position 0 up to the array position i. The data structure operations are subscripted with the abbreviation of the corresponding data structure name, so that, for example, the `Insert` operation is denoted by $\texttt{Insert}_{\texttt{PT}}$ in case we mention the Patricia Trie's `Insert` operation.

2.3 Models

After defining the used notation and the problems to be solved, we present the models used for solving those problems. It is important to define the model of computation and the alphabet model in order to analyze and predict the running time of the algorithms.

An alphabet model can be defined upon a totally ordered alphabet Σ from which the string characters are taken. Worst-case space complexity and string comparison time complexity are also part of its definition. It defines the data type of the elements being processed, in terms of string representation and the internal memory computation, impacting directly the analysis of the algorithms and the data structures. For instance, it can define how many elements can be compared during a single step of CPU processing.

The models of computation play an important role, because they define how accurate we can predict the execution time of the string processing algorithms in different computer archi-

tectures, approaching different aspects. Our algorithms are based on the Disk Access Model (DAM), which is mentioned below.

We define and contextualize four different alphabets below, in order to show one way of how theory could meet practice in algorithm engineering.

Definition 2.1. *(**Integer Alphabet**) The characters are the integers out of the range {1, ..., N} and each character occupies one single machine word, such that a character uses $\mathcal{O}(1)$ space. All usual integer operations can be performed in $\mathcal{O}(1)$ time on the characters.*

The Integer Alphabet encompasses the most convenient situation where each string character fits exactly into one memory cell, so that it can be used in combination with simple models of computation (e.g. RAM). It is well known, for instance, that the fastest algorithm for building STs of strings over an integer alphabet out from a *polynomial range* runs in $\mathcal{O}(n)$ time [27], where n is the size of the text.

Definition 2.2. *(**Constant Alphabet**) The characters are drawn out of the range {1, ..., N}, but this model excludes the condition that a character can be singly stored in a machine word, so that space complexity is $\mathcal{O}(n)$, where n is the text size. Whenever performing dictionary operations $\mathcal{O}(1)$ internal time is required.*

The Constant Alphabet assumes that the characters are bigger than the memory cells. The running time lower bound for ST construction in the comparison-model is $\Omega(n \log n)$ [27]. For example, this model could be useful whenever designing algorithms for a computer architecture with 16 bits processors, which are used for indexing strings over an alphabet where the characters have more than 16 bits.

Definition 2.3. *(**Packed String Alphabet**) Multiple characters can be represented using a single machine word. The characters are the integers in the range {1, ..., $|\Sigma|$}, where $|\Sigma| \leq N$ and N is the total number of characters of a string. Each machine word contains $\Theta(\log_{|\Sigma|} N)$ characters and the amortized comparison time is $\mathcal{O}(1/(\log_{|\Sigma|} N)$. Absolute space and time complexities are $\mathcal{O}(1)$.*

The Packed String Alphabet (3) captures the situation where the alphabet characters are smaller than the memory cell (e.g. characters representing DNA sequences encoded as `00`, `01`, `10` and `11`). Regarding the overall size, the size of the text n is given in number of machine words, i.e. $n = \Theta(N/\log_{|\Sigma|} N)$, so that the size of the text is $\Theta(n)$ machine words and the size of the full-text index is $\Theta(N)$ machine words [21].

One further definition could be the most general definition of an alphabet, where the strings are drawn over an infinite alphabet (e.g. all natural numbers). In this sense, the time and space complexities are dependent on the algorithm state and can't be directly defined. It would express the definition for problems where no assumptions about the data type are done, so that any auxiliary model can be used for modelling computation in exchange for a less accurate complexity analysis.

After defining the data being processed and its atomic costs, the used computation model is briefly explained, namely the DAM. In this model, the memory hierarchy is composed of multiple levels with different speed and characteristics, being the simplest suitable model for external memory full-text indexes. In its simplest version, a central processing unit (CPU) is used to process the information, while two layers of memory are used to store the data being processed. The memory level which is closer to the CPU is smaller and faster, as the memory level which is farthest from the processor is slower and bigger. The DAM [38] has a set of parameters (N, M and B), where:

- N: number of characters in the text to be stored in secondary memory
- M: number of characters that fit into internal memory
- B: number of characters that fit into a disk block (number of elements transfered between two levels of memory)

Using the model mentioned above, a large number of external memory algorithms can be assembled with the following three ingredients: scanning, sorting, and searching. These shorthands are sufficient for analyzing the algorithms in the scope of this thesis and the proofs of the upper and lower bounds for the number of I/Os needed to perform the single operations can be found in [38]. They are short for:

- $\text{scan}(N) = \Theta(N/B)$ I/O
- $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ I/O
- $\text{search}(N) = \Theta(\log_B^N)$ I/O

Scanning operates sequentially on the data blocks to be processed. Sorting firstly sorts the blocks to be accessed by their addresses, in order to achieve sequential accesses - their addresses reflect their physical disposal on the external memory device. Searching is based on the pointer-model, where a block is visited if and only if another block references it.

2.4 B$^+$-Trees

A B-Tree is one possibility to realize an associative array mapping keys to values and generalizes the binaray search trees [46], namely the number of children of the nodes can be greater than 2, so that more than two comparisons are required in order to determine the lexicographic position of a query at a given tree node. B denotes the branching factor, i.e. the number of children of a node, and that parameter B can be used as both tuning parameter or physical magnitude. Making B larger reduces the number of performed I/O operations, but enlarges the volume of the I/O content. Whenever dealing with complex architectures, this parameter is not a physical magnitude anymore and must be determined upon a myriad of factors.

Given this data structure organization, we perform search from the root node towards the leaves and a B-way decision algorithm must be implemented for choosing the next child node to be used for continuing search. The node can be simply implemented as a sorted sequence of *fixed-length* keys for later binary search or, for example, as a trie data structure supporting *variable-length* keys. The former solves the predecessor problem in $\mathcal{O}(\log B)$ time meanwhile requiring fixed-length keys, as the latter does it in $\mathcal{O}(p \log \sigma)$ and provides the flexibility of using variable-length keys at cost of extra space consumption and being input-sensitive regarding running time.

This data structure aims to be balanced on the number of nodes per level, in order to keep the tree's height balanced and minimize the number of I/O operations. Unfortunatelly, keeping the balance under dynamic operations is an NP-complete problem, so that heuristic algorithms must be applied to guarantee good running times.

A commonly used variant is the *B^+-Tree*, which stores all the satellite information in the leaves and only keep keys and child pointers in the internal nodes. Requiring the internal node to be at least 2/3 full prior to splitting, rather than at least 1/2 full, yields another variant: the B*-tree.

Definition

In the following, we assume that the values (satellite data) are directly stored in the leaves. Alternatively, one could simply store a pointer to those data, adding an extra indirection level,

but yielding better locality.

Our definitions are based on the ones provided by Cormen [46], but we go directly to the definition of the B+-Tree.

Definition 2.4. ***(B+-Tree)*** *A B+-Tree is a rooted ordered tree with the following properties:*

1. *(content) Every node u holds the following data:*
   - *(i) the number of keys stored in the node (node occupation)*
   - *(ii) the keys themselves*
   - *(iii) the child pointers*
2. *(structure) The keys $key_i[u]$ of a node u separate the key ranges stored in each subtree $children_i[u]$, so that any key smaller or equal than $key_i[u]$ resides in $children_i[u]$ for $0 \leq i \leq B$, and any key strictly greater than $key_B[u]$ resides in $children_{B+1}[u]$.*

   The following invariant must hold, if we pose a query $\boldsymbol{k}_j \in keys[children_j[u]]$ at node u:

   $$\boldsymbol{k}_1 \leq key_1[u] \leq \boldsymbol{k}_2 \leq key_2[u] \leq ... \leq key_B[u] < \boldsymbol{k}_{B+1}$$
3. *(structure) All leaves have the same depth, i.e. the tree's height h.*
4. *(configuration) Lower and upper bounds are imposed for the internal node occupation, expressed as t: $t - 1 \leq |keys[x]| \leq 2t - 1 = B$. A node with $2t - 1$ keys is said to be overflowing and a node with $t - 1$ keys is underflowing, t is the degree of the B-Tree. Exceptionally, the root can have at least one key and two children.*

With Property 1 we can precisily define the memory consumption for the index and estimate the best value for B based on the node organization. Property 2 shows the requirements to be fulfilled by the `Search` operation. Indirectly, it also defines that a given key-separator appears olny once along the whole index. Property 3 suggests that the execution time in terms of I/O operations is precised by the tree's height, accounting on a predictable performance based on the worst-case execution time. Property 4 establishes the mechanics for the dynamic operations in terms of the underflowing/overflowing situations and provides the needed definitions for calculating the tree's height. The next figure represents the structure of a B-Tree with $B = 4$.

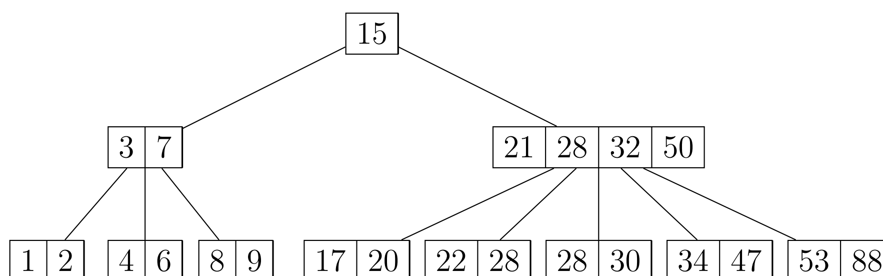


Figure 1: B-Tree example

Observe that the tree's height is not determined considering the key contents, but the tree structure allows for a better organization of its representation. The next theorem formalizes this result.

Theorem 2.5. *Given a tree designed for holding n key-value pairs with minimum degree t, the tree's height can be precised: $h \leq \log_t(\frac{n+1}{2})$*

The proof of the Theorem 2.5 uses basic combinatorics and is referred to Cormen [46]. Nevertheless, this result is very important for enumerating the time complexity of the B+-Tree operations. All operations `Search`, `Insert` and `Delete` have optimal worst-case performance of $\mathcal{O}(h)$ I/O operations in the pointer-based model [38].

Applicability of B$^+$-Trees

The main memory of a computer consists of silicon memory chips and are rather fast, but also expensive. On the other side, other storage devices based on magnetic disks (or think about the network model) are typically two orders of magnitude cheaper per stored bit than silicon-based chips, but are rather slower.

Data structures like the B-Tree and its variants aim to cope with those situations, whenever balancing costs and performance is an issue, mainly in systems with different layers of memory. It comes out that data is kept sorted in order to perform sequential accesses exploiting locality. Furthermore the data structure is designed to provide single accesses to larger blocks of data, in order to amortize I/O costs.

A simple physical description of a typical secondary memory device, namely the hard disk, can be found in [46] and the main result is that the costs of the I/O operations can be reduced in an amortized sense, by using the time spent waiting for mechanical movements to transfer a larger volume of data, rather than transferring single elements.

2.5 Basic Text Indexes

There is a basic toolbox for full-text indexes composing the basis for all their variants. These data structures are very popular, but recommended to be used only for solving problem instances fitting in main memory. We sketch the main points of this basic toolbox, namely we briefly explain the tries, STs and SAs. The conversion between the two latter data structures are well-known procedures and we approach them in this thesis in order to solve the problem of constructing the String B-Tree efficiently. Furthermore tries are the principal building block of our index.

Tries and their variants are elegant solutions that can be used to realize an associative array with variable length keys. The ST [30] is a powerful data structure in terms of its operations for solving combinatorial problems, but it consumes very much space for the structural pointers and has bad locality regarding the `Search` operation.

SAs overcome the space consumption problem at the cost of slower operations, but still don't have good locality for searching. Whenever combined with the LCP array, the SA directly represents the ST without extra pointers, being compact and still suitable for faster searches.

Among other approaches, DAWGs (Directed Acyclic Word Graphs) turn out to be also a starting point for designing data structures to solve the problems presented in this thesis [1]. Their major problem lie on the lack of flexibility in the dynamic scenarios and the lack of regularity for memory accesses. Thus, they are barely discussed in this thesis.

2.5.1 Tries

A trie is a dictionary data structure for keys of any data type and variable length. In order to support keys of variable length the internal nodes can not directly represent the keys, as it happens with other key-value data structures (e.g. binary search trees).

The prefix shared between keys is expressed via internal nodes with more than one descendent. A bifurcation at a node distinguishes two different keys represented by the trie. The leaves hold the values corresponding to the keys, that result from the concatatenation of all edge labels from the root node down to that leaf. Next, the tries are formally defined.

Definition 2.6. ***(Trie, Prefix Tree)*** *A trie is an ordered multiway tree, where the keys have variable length and are unique. The tree edges are labeled with digits coming from the keys and the leaves identify the complete keys.*

The Figure 2 is an example of a trie holding the key-value pairs from $\mathcal{X} = \{[\texttt{abcd}, 1E3], [\texttt{abc}, 3B6], [\texttt{cbc}, UB7]\ [\texttt{cpc}, L91]\}$.

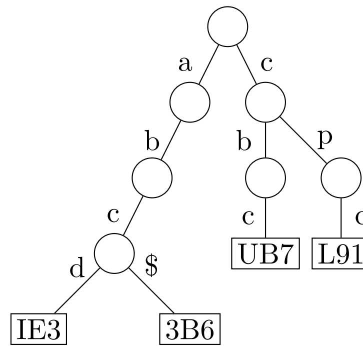


Figure 2: Trie example

We can observe that the Definition 2.6 yields a data structure with very bad space-usage, because each character of the keys have a corresponding internal node, which induces edges. Since the number of edges is $\mathcal{O}(N)$, the total number of symbols to label the tree is the total number of symbols for labeling the edges, bounded by $\mathcal{O}(N)$, plus $\mathcal{O}(|\mathcal{S}|)$ symbols for labeling the leaves. In this way, the oveall space consumption for labeling the tree is $\mathcal{O}(N \log N) + \mathcal{O}(|\mathcal{S}| \log |\mathcal{S}|)$.

In the worst-case all digits of all keys are compared to the pattern, resulting on $\mathcal{O}(N)$ character comparisons to find the corresponding node, plus Z comparisons to report the descending occurance keys. Notice that $\mathcal{O}(N)$ is in $\mathcal{O}(|\mathcal{S}|^2)$.

The next variant eliminates nodes with only one descendent in order to improve running time in avoiding unnecessary comparisons.

Definition 2.7. ***(Compact Trie, Compact Prefix Tree)*** *A compact trie is trie, where each node with a single child is merged with its parent.*

The Figure 3 shows an example of a compact trie constructed for the same set represented in the trie of the Figure 2.

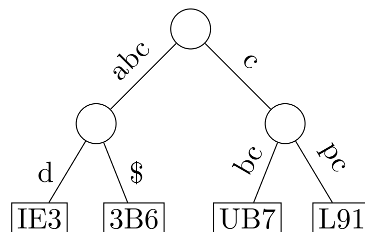


Figure 3: Compact Trie example

Eliminating nodes with single descendents reduces the number of edges, but it changes character-based edge labeling for substring-based edge labeling. Thus, the number of edges of the trie is reduced from $\mathcal{O}(|\mathcal{S}|^2)$ to $\mathcal{O}(|\mathcal{S}|)$, but the space used for the labels is the same. At the bottom line, the space consumption for the tree pointers is improved, as the number of edges decreases, but the space used for labeling the trie and the number of comparisons are the same.

Given the simpler data structures above, one still would like to compress it and improve running time. A simple trick, not to store the edge labels directly along the data structure, is to store a

pair of pointers to the begin and end of those substrings. Such an approach with text pointers is not suitable for external memory algorithms, because every substring comparison induces random accesses.

In order to elaborate a more efficient data structure, only the first branching characters of each edge and the substring size are stored. Edges turn out to consume size bounded by an alphabet character plus a symbol coming from a reduced universe, which represents the edge's substring size. The later symbol doesn't involve the memory address space, what is a great improvement against the trick mentioned above.

By doing so the a single *membership* query costs only one text access and a *predecessor* query costs only two text accesses. Although PTs are lossy compressed, we observe that it turns out to be a data strucutre which is very suitable for external memory models.

The following definition formalizes the data structure described in the previous paragraph and is similar to the definition proposed by Ferragina [34].

Definition 2.8. ***(Patricia Trie)*** *[11, 34] A Patricia Trie* $\mathtt{PT}_\mathcal{S}$ *built on* $\mathcal{S}$ *satisfies the following conditions:*

1. *Edges are labeled by one branching character from* Σ*. Internal nodes have at least two outgoing edges labeled with different characters. Edges of an internal node are ordered according to* $\leq_L$*.*
2. *Each string from* $\mathcal{S}$ *has a distinct leaf.*
3. *Every lowest common ancestor (LCA)* u *of two leaves* a *and* b *is labeled by an integer* $len(u) = \mathtt{LCP}(string(a), string(b))$*.*

Fact 2.9. *Since only the root node can have degree less or equal 1, a Patricia Trie with* $|\mathcal{S}|$ *leaves has no more than* $|\mathcal{S}|$ *internal nodes.*

Since only the branching characters are stored and the total number of internal nodes is bounded by $\mathcal{O}(|\mathcal{S}|)$, the overall size used for encoding the internal node and edge labels is $\mathcal{O}(|\mathcal{S}| \log N)$.

With the new features for routing the `Search` operation introduced by the PTs, we must deal with inexact results for the *precessor problem*, but the results can be quickly checked with an additional verification step, which carries out the same worst case of the $\mathtt{Search}_\mathtt{PT}$ operation plus the $\mathcal{O}(p)$ for calculating the LCP value between query and found string.

Another interesting result is that binary encoded PTs have exactly $|\mathcal{S}| - 1$ internal nodes and don't require storing the any label. More details of the PT are contextualized in the section 3 and the Figure 4 shows an example of a PT constructed for the same set represented in the trie of the Figure 2.

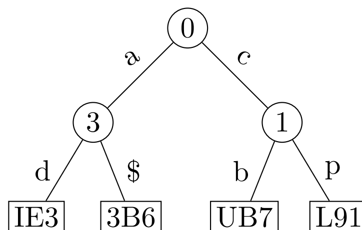


Figure 4: Patricia Trie example

2.5.2 Suffix Arrays

A SA $\texttt{SA}_{\mathcal{T}}$ is the lexicographically sorted array of suffixes of a text $\mathcal{T}$. The suffixes sharing a given prefix always form a contiguous interval. By binary searching the array, a string matching query can be answered with $\mathcal{O}(\log_2 N)$ string comparisons, requiring $\mathcal{O}(|P| \log_2 N)$ string comparisons in the worst case, when each character of the pattern triggers a mismatch. In the following picture we show an string (left) and its corresponding SA (right):

$\mathcal{T} =$

a	t	a	j	l	t	u	x	e	b
1	2	3	4	5	6	7	8	9	10

$SA_{\mathcal{T}} =$

2	0	9	8	3	4	1	5	6	7
1	2	3	4	5	6	7	8	9	10

Figure 5: Suffix Array example

Storing extra information along the SA, for example the array of LCPs (longest common prefixes), allows for the reduction of search time from $\mathcal{O}(|P| \log_2 N)$ to $\mathcal{O}(|P| + \log_2 N)$. Using this extra data eliminates unnecessary comparisons during the binary search, more exactly one only compares $P[\texttt{LCP}(i) + 1]$ with the $\mathcal{T}[\texttt{SA}[i] + \texttt{LCP}[i] + 1]$, in order to decide whether to continue the search in the left or right sub-range of the `SA`. The `LCP` array works like an oracle with precalculated references to the exact characters to be compared during the binary search, allowing for the implementation of a predictable `Serach` operation.

2.5.3 Suffix Trees

The ST is another versatile data structure that is fundamental to string processing applications and can be constructed by inserting all suffixes of s into a compact trie. It relies on preprocessing the text in order to answer substring queries with fast running times. It also has fast search times, because it deeply exposes the internal structure of a string [7]. The next definition formalizes this data structure.

Definition 2.10. ***(Suffix Tree)*** *The suffix tree of a string s is the compact trie built on all n suffixes of s.*

In order to ensure that no suffix is a proper prefix of another suffix, we add a further termination symbol to the end of the text. That sentinel character, say '\$', doesn't appear anywhere else in the string s and is lexicographically smaller than any other symbol in Σ. This has the implication that each suffix of s has an *unique leaf* in the ST, since any two suffixes will follow separate branches in the ST, meaning that the set of suffixes is prefix free.

We precise some further aspects of the ST that will be used in rest of the text. For a given node u, its *path label* is defined as the concatenation of edge labels on the path from the root to u. The *string depth* of a node u is simply the length of the path label. For any two suffixes s_i and s_j, let the string t be their *longest common prefix* (LCP), then there necessarily exists a node u whose path label equals t. That node u is the so-called *longest common ancestor* (LCA) of both leaves i and j representing s_i and s_j. The Figure 6 shows the ST for the same input string $s = \{\texttt{BANANA\$}\}$.

The trick for representing the edge labels mentioned in the subsection of the compact tries is crucial so that linear space representation of the ST is possible. Note that in the bit-complexity model the ST can be stored in $\Theta(n \log n)$ bits, whereas the text needs $n\lceil \log \sigma \rceil$ bits.

Clearly that space complexity is not optimal for any σ, since σ^n different words (or maps) over Σ can be generated. On the other side n^n different elements require $n \log n$ bits to be represented and we are often given that $\sigma \ll n$ [44].

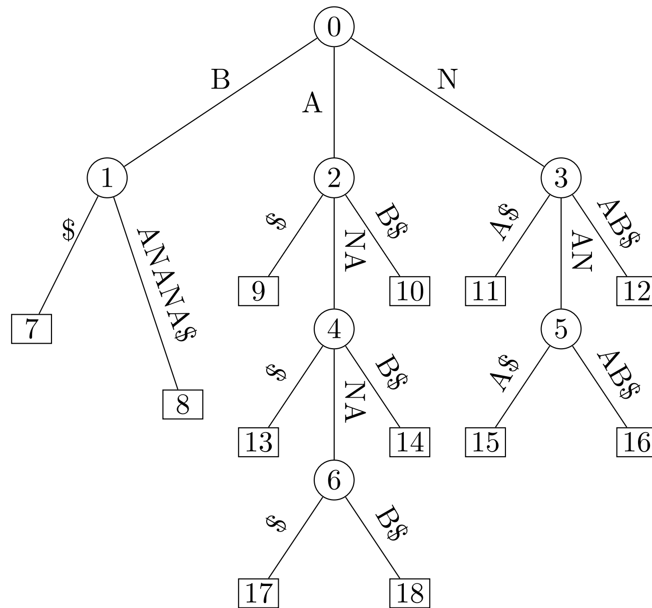


Figure 6: Suffix Tree example

Construction algorithms are mostly designed in the constant alphabet model [13, 30], but the algorithm in [26] is also analyzed with the integer alphabet model. Fast construction algorithms rely on *suffix links*, a special pointer connecting a node representing a string $s\alpha$ to the node representing α, where $s \in \Sigma^*$ and $\alpha \in \Sigma$. This is something we don't use in our algorithms, making our solution faster and simpler, at the cost of requiring the precalculation of the SA and LCP arrays.

Complexity Summary

The integer alphabet model is used for analysis and the space consumption for pointers is not regarded, because we focus on space-efficient tree representations in this thesis. If considering the tree pointers, the number of extra pointers is bounded by the number of tree edges. In that case, tries carry $\mathcal{O}(|\mathcal{S}|^2 \log |\mathcal{S}|^2)$ extra bits, as compact tries and Patricia Tries carry $\mathcal{O}(|\mathcal{S}| \log |\mathcal{S}|)$ extra bits. The analysis presented here is enough to show the evolution of the data structures and to point out their advantages and disadvantages regarding the external memory model.

The algorithms studied here are also output sensitive, meaning that the number of answers Z matters. We abuse that in real world scenarios the following equality holds: $|\Sigma| \ll |\mathcal{S}| \ll N$. Regarding the space complixity terms presented in the Table 2.5.3, the first part denotes the amount of bits used to encode the edge/node labels and the second to encode the leaf labels.

In comparing tries, compact tries and PTs, all of them have time complexity of $\mathcal{O}(p \log \sigma + Z)$ for the search operation, except for the PTs that don't use all the p pattern characters for comparison during searching. Thus, we should provide a more detailed analysis of the `Search` operation performance based on the trie's height - the number of edges of the trie in the worst-case - and not the pattern length. Observe, that tries and compact tries are content-sensitive as PTs are not.

Data Structure	Index Space (bits)	Search (comparisons)
Trie	$\mathcal{O}(N \log N) + \mathcal{O}(\|\mathcal{S}\| \log \|\mathcal{S}\|)$	$\mathcal{O}(N + Z)$
Compact Trie	$\mathcal{O}(N \log N) + \mathcal{O}(\|\mathcal{S}\| \log \|\mathcal{S}\|)$	$\mathcal{O}(N + Z)$
Patricia Trie	$\mathcal{O}(\|\mathcal{S}\| \log N) + \mathcal{O}(\|\mathcal{S}\| \log \|\mathcal{S}\|)$	$\mathcal{O}(\|\mathcal{S}\| + Z)$
Suffix Array	$\mathcal{O}(\|\mathcal{S}\|)$	$\mathcal{O}(p \log \|\mathcal{S}\| + Z)$*
Suffix Tree	$\mathcal{O}(N \log N) + \mathcal{O}(\|\mathcal{S}\| \log \|\mathcal{S}\|)$	$\mathcal{O}(N + Z)$

Table 1: Worst-case summary table for the basic indexes

* String comparisons, since search is done without the LCP array

2.6 External Memory Text Indexes

I/O efficient full-text indexing data structures can be represented by the Hierarchies of Indexes [41], Compact Pat Tries [5] and the String B-Trees [37].

2.6.1 Hierarchies of Indexes using Suffix Arrays

The basic idea is to build a *supra-index* on the top of the SA, composed of shorter SAs. Theoretically this only provides performance improvement by a constant factor. However, by better exploiting the memory cache levels one can show that those improvements are rather bigger via experiment results in real machines.

The SA is firstly divided into blocks of size p and we hold one SA element of each block in main memory together with the first l characters of the corresponding suffix. `Search` is done in two steps: first, the supra-index is binary searched in order to find the SA block where the occurances lie. Afterwards, the block is further binary searched using the l characters cached from the text. Both steps solely access main memory cells, thus disk accesses occur only in case the longest common prefix between query and the string entries are greater than l.

As a result, it turns out to be an efficient cache-scheme designed for SAs and parametrized with p and l. Instead of doing $\mathcal{O}(|P| \log |\mathcal{S}|)$ disk accesses, only $\mathcal{O}(\log |P|)$ disk accesses are done. Further it turns out to be also very space-efficient, cosuming the same space of the classical SAs plus the size of the supra-index.

Although space-efficient and alphabet-independent, the SA is inherently static and the number of disk accesses is not optimal by the addtive term $\mathcal{O}(\log_2 \frac{N}{B})$, because of binary search. The dynamic SA can be adapted to work in secondary storage, requiring $\mathcal{O}(\frac{N \log_2 N}{B})$ extra space [31].

2.6.2 Compact Suffix Trees

This data structure offers a compact representation and is adapted to external memory to reduce the number of disk accesses while searching. Dynamic operations are also provided, in contrast to the Hierarchies of Indexes, but they degenerate search performance and space usage as they are executed. The basic idea is to partition the Suffix Tree into pieces that fit into a disk block, so that no disk accesses are required whenever searching within a single block.

The compact representation relies on using binary encoding of the characters, getting rid of the space needed for storing the edge labels. Thus, the comapct tree encoding can be simply stored using a succint representation based one the Jacobson approach.

The strategy for partitioning the Pat Tree is based on a greedy bottom-up algorithm that minimizes the maximum number of disk accesses, but yields small pages and poor fill ratios. Unfortunately, comparing the edges' labels after a branch requires further accesses to the pages containing the corresponding substring and it makes the page accesses irregular. Searching for P might visit $\mathcal{O}(P)$ nodes in distinct pages and the occurances found might be addtionally stored in $\mathcal{O}(Z)$ other pages.

The search time is distribution-dependent because it depends on the tree's height $H = \Theta(N)$ in the worst-case. Based on this assumption, they show that the maximum number of pages traversed on any root to leaf path is at most $1 + \lceil H/\sqrt{B} \rceil + \lceil 2 \log_B N \rceil$, thus search time costs $\mathcal{O}(\text{scan}(|P| + A) + \text{search}(N))$ I/O steps, assuming that $H = \mathcal{O}(\sqrt{B} \log_B N)$ [21].

Apart from the theoretical drawbacks cited above, the suffix links might be updates after having been set. The number of suffix links can be huge for big alphabets, namely $\mathcal{O}(|\Sigma|)$ incoming suffix links can point to the same node and must be properly managed under string insertion and removal.

2.6.3 String B-Tree

This data structure definitively links external memory data structures to string matching data structures and can be indirectly understood as a supra-index for the SAs and LCP arrays. Even though it has been called String B-Tree, it is a B^+-Tree where the keys are the (suffixes of the) strings from $\mathcal{S}$ in lexicographical order.

It overcomes the limitations of *inverted files*, which are not dynamic and are restrictively word-based. It deals with the drawbacks of the *SAs*, that also have an impairment on modifiability as well as problems with locality. The unbalanced tree structure of the *STs* is ameliorated by referring to the children during search using a B-Tree structure, rather than greedily partitioning trees.

The String B-Tree has the same worst-case performance of the B-Trees, being optimal in the pointer-based search model regarding the $\texttt{Search}_{\texttt{SB}}$ operation [38]. Update operations, like $\texttt{Insert}_{\texttt{SB}}$ and $\texttt{Delete}_{\texttt{SB}}$, have fast execution times via the conventional B-Tree heuristics [12].

The strength of the String B-Tree is the ability of performing powerful combinatorial search operations, as the ones supported by the STs, meanwhile making no assumptions upon the input distribution whenever referring to the $\texttt{Search}_{\texttt{SB}}$ operation. It implies better worst-case guarantees, than the ones of the Compact Suffix Trees for external memory.

By using the *blind trie* data structure, each node stores $\mathcal{O}(B)$ characters and pointers to strings. The stored information within a node allows for algorithms that are independent of the total length of the pointed strings, minimizing the number of page acceses.

Since keys are the logical pointers to the strings and the order between the keys is the lexicographic order among the strings pointed by them, the data structure is able to handle strings with unbounded lengths [34].

Each String B-Tree's node v is stored in a disk block and contains an ordered string set $\mathcal{S}_v \subseteq \mathcal{S}$, such that $b \leq |\mathcal{S}_v| \leq 2b$, where $b = \Theta(B)$ and can be used as tuning parameter. One represents those keys as a Patricia Trie, a.k.a. Blind Trie, because of its features. The branching factor of the String B-Tree is $\Theta(B)$ and the consequence of that is the worst-case tree's height is $\Theta(\log_B N)$, optimal among all variants.

At the internal nodes, we search a child node u whose interval $|L(u), R(u)|$ contains P and this is done via the $\texttt{Search}_{\texttt{PT}}$ operation, requiring a single I/O step and $O(|P| \log \sigma)$ time for internal work. In fact, whenever implementing the String B-Tree those ranges are not explicitly used.

Complementary Techniques

There are two other techniques that can be combined with the data structures above. *Binary encoding* of the strings can be used to supress all the tree labels, accounting on significant less space usage, depending on the symbol distribution. However, it worths to notice that the emerging binary trees can have longer heights, namely the height of such encoded trees is bounded on the number of bits of the largest key, and that can imply much more comparisons done during a single operation, since single-child nodes are supported. In fast storage systems and fast networks, data transfer costs are low, so that reducing the number of comparisons turns out to be more relevant. Also, in this thesis we are interested in evaluating the theoretical worst-case performance of the algorithms without considering entropy compression. A data structure for representing the internal nodes with height in $\mathcal{O}(|\mathcal{S}|)$, rather than in $\mathcal{O}(N)$, seems to worth much more and we don't consider this technique in our implementations.

Lexicographical naming is a second method for reducing the index size, meanwhile also improving the worst-case execution time of the index operations. A map $\alpha : \mathcal{S} \rightarrow \{0, \ldots, |\mathcal{S}| - 1\}$ is called lexicographical naming if the following holds: for $X, Y \in \mathcal{S}$, $\mathrm{X} <_L \mathrm{Y} \implies \alpha(X) < \alpha(Y)$. If the set of strings is static, order preserving minimal perfect hashing functions (MPHF) can be applied so that switching alphabets costs $\mathcal{O}(1)$ execution time. The applicability of lexicographical naming depends on the problem requirements, i.e. whether the query answers comprise listing whole strings, specific string fragments or any substring taken out from Σ^*. We target the evaluation of the solutions for the latter case, so don't consider this technique for the implementation of our algorithms.

Complexity Summary

In the space usage analysis of the Hierarchie of Indexes, p denotes the size of the shorter SAs and l denotes the number of characters hold in main memory per block.

Observe that we only compare space usage and search performance, whenever comapring the guarantees of the String B-Tree with any other external memory data structure. Considering the dynamic operations would be something unfairly degenerative in any case.

Data Structure	**Index Space** (bits)	**Search** (I/Os)	**Tree Height**
Hierarchie of Indexes	$\mathcal{O}(\lvert\mathcal{S}\rvert + (1 + l)\frac{N}{p})$	$\mathcal{O}(\lvert P\rvert \log N)$	-
Compact Pat Trees	$\mathcal{O}(N)$	$\mathcal{O}(\text{scan}(\lvert P\rvert) + \sqrt{B} \log_B \lvert\mathcal{S}\rvert)$	$\mathcal{O}(\sqrt{B} \log_B \lvert\mathcal{S}\rvert)$
EM Patricia Trie	$\mathcal{O}(\lvert\mathcal{S}\rvert)$	$\mathcal{O}(\frac{\lvert S\rvert}{\sqrt{\lvert P\rvert}} + \log_{\lvert P\rvert} \lvert\mathcal{S}\rvert)$	$\mathcal{O}(\lvert\mathcal{S}\rvert)$
String B-Trees	$\mathcal{O}(\lvert\mathcal{S}\rvert)$	$\mathcal{O}(\frac{\lvert P\rvert}{B} + \log_B \lvert\mathcal{S}\rvert)$	$\mathcal{O}(\log_B \lvert\mathcal{S}\rvert)$

Table 2: Worst-case summary table for the external memory indexes

We can observe that the String B-Tree is the best theoretical deal whenever simultaneously considering (i) the constants involved in the worst-case space consumption, (ii) worst-case execution time of `Search` and (iii) the implementation of dynamic operations.

Note that the String B-Tree is composed of a set of PTs resulting of splitting a big EM Patricia Trie. It means that we can even save a bit of space, because some internal nodes get merged with their parents after splitting. Thus, in practice the constants in $\mathcal{O}(|\mathcal{S}|)$ of the String B-Tree are smaller than the ones of the EM Patricia Trie.

2.7 More Related Work

Efficient parallel and distributed external memory approaches for full-text indexing use mostly SAs and STs. However, as mentioned in the introduction, the major drawbacks are the expensive construction times, because of the inherent problem difficulty, and the costful search operation, because of the bad locality of full-text indexes.

In order to deal with construction slowness, some algorithms have been proposed for construction of SAs [49, 22, 17, 47, 16, 19, 9] and construction of STs [26, 45, 27, 23, 4, 25, 15, 29, 40, 39]. However, searching in a SA still can get much slower in huge text databases, because the search operation relies on binary searching the suffixes and compares each suffix during each search step. Reconstructing the ST from the SA is a first improvement of for the search operation.

Beside the SA, further data structures (e.g. STs, PTs, among others) are recommended to be built over the SA in order to speedup search at cost of extra memory consumption. The efficient construction algorithms for the STs, one of the most powerful data structures used in string processing, don't mention friendly worst-case analysis for the search operation. In this way access patterns are non-predictable, accounting for varying performance rates with different text sizes. Those construction algorithms are mostly based on *paged* approaches or simply rely on the rules of the operating system, being mostly decorated with *multithreading* in order to deceivably present more advanced algorithms.

The first description of the String B-Tree data structure was published in [32], followed by the experimental studies [33] and a more detailed description in [34]. From there on, some studies had been accomplished regarding the performance and applicability of the data structure, but still no uniform study has been presented which (i) deals with the elementary problems in the data structure implementation, like the internal node design, (ii) present fair comparisons with the concurrent approaches in terms of both construction and in-operation performance.

After publication [32], one year later the first experimental study had been published [33], confirming the theoretical predictions that searches are fast, since $\log_B^N$ is much smaller than $\log_2^N$, where N is the input text size and B is the B-Tree block size. The comparison had been done with an SA implementation and a prefix B-Tree of the UNIX operating system. At that time, no external memory implementation of a ST was available and no benchmarking ST implementation had been developed. At that time, STs in external memory were not something simple to develop, so a face-to-face comparison with the state-of-the-art was not provided. The empirical counting of disk access was used as performance metric and the resulting numbers make sure that the String B-Tree is a strong competitor for the SA in external memory because of *regular* disk access patterns.

The Master Thesis of Manuel Scholz [24] is an interesting experimental evaluation of the data structure followed by a neat description of the elementary pieces. The `Search` operation is the starting point for analysis of the data structure properties. The influence quantities, namely the tree size and node size, are contextualized with their impacts, as well the role of the text size. The validation and comparison is done via comparison with a ST implementation with unknown origin and the metrics used in the analysis are the number of disk access as well as CPU processing time. At the end an analysis of the practical usage of the String B-Tree is presented based on the estimation of its properties, but still experiments are not performed with huge input sizes and the implementation "packs" and "unpacks" the internal nodes every time they are unloaded/loaded.

3 String B-Tree

The object of construction of this Bachelor Thesis is the String B-Tree data structure firstly proposed by Ferragina [32]. We use the definitions and theorems given in a later paper [34], while adapting some details of the definition for sake of efficiency. This adaptation is reasonable in terms of space and time efficiency and had been also adopted in the Master Thesis of Manuel Scholz [24], namely we don't store lower and upper bound strings within the internal nodes, but only the upper bound strings. It doesn't affect correctness, because the string ranges are likewise well-defined and still let the algorithms, implementations and experimental evaluations of [32, 34, 34, 50, 35, 36] be compatible and comparable.

This data structure seems to combine the power of theory with the craft of praxis. One of its main advantages is the ability of handling keys with arbitrary lengths (strings), offering a good solution for pattern matching in big databases with data of different types. It also loads data blocks on demand, offering very a good locality. The former point comes from the usage of PTs for representing the B-Tree internal nodes and the latter comes from the fact that B-Trees are balanced trees with optimal search time in the pointer model [38].

The original paper exposes how the B-Tree data structure combined with the PTs doesn't degenerate searching with the index. In our work, we propose a space-efficient variant of the String B-Tree and show an efficient method of constructing it.

3.1 Definitions

The following definitions help to precise the output of some atomic operations defined lately.

Definition 3.1. ***(Hit Node*** *h**)*** *[34] Given a Patricia Trie* PT *and pair (l,k), where l is a leaf and $0 < k \leq$ length(l) is a natural number, the hit node h is an ancestor of l satisfying: length(h) $\geq k >$ length(parent(h)). If $k = 0$, the hit node is the root.*

The hit node defines the outcomes of the $\texttt{BlindSearch}_{\mathsf{PT}}$ operation in terms of the node at which the algorithm stops and this operation is defined later in the subsection of the extended PT.

Definition 3.2. ***(String Position*** *j**)*** *Let $\mathcal{S}$ be a set of strings drawn using an alphabet Σ and $s \in \Sigma^*$ be any string. The position j of s in the set $\mathcal{S}$ is the number of strings which are lexicographically smaller than s in $\mathcal{S}$.*

The string position is the main information to be used for routing serach along the B^+-Tree and is given by the $\texttt{GetPosition}_{\mathsf{PT}}$ operation. Before diving into its details, we briefly explain the $\texttt{Search}_{\mathsf{SB}}$ and $\texttt{Insert}_{\mathsf{SB}}$ operations in the next subsections.

3.2 $\texttt{Search}_{\mathsf{SB}}$ Operation

We analyze the String B-Tree algorithms using the DAM model, that is in each step of the $\texttt{Search}_{\mathsf{SB}}$ operation we load a node, use it to choose a child node for continuing search and directly erase it from main memory. In this sense, we don't give any special attention for runtime stack usage as usually done in RAM algorithms. So, we adopt the more elegant recursive version of the search algorithm and don't use the iterative version, which is less readable.

Observe that the recursion depth of this algorithm never exceeds $\mathcal{O}(\log_B N)$, the tree's height. $\texttt{GetPosition}_{\mathsf{PT}}$ solves the predecessor problem with a PT, that represents a range of key

Algorithm 1: `Search`$_{\texttt{PT}}$ operation

```
Input: Query P, Root Node root
1 if root.IsLeaf() then begin
2     return root.BinarySearch(P)            // similar complexity as if done with SAs
3 else begin
4     pos ← root.GetPosition_PT(P)
5     child ← root.GetChild_PT(pos)
6     return Search_SB(P, child)             // at most O(log_B N) recursive calls
```

separators (string separators). *Binary search* can be used instead of *linear search* for searching within the leaves (see line 2 of the Algorithm 6), because the strings stored at the leaves are lexicographically sorted. Moreover, it can reuse the LCP array to speed up searching and the complexity of this step is based on the size of the leaves, not the total text size, being much more efficient as if done solely with the SAs.

The `Search`$_{\texttt{SB}}$ procedure enables us to solve the external memory substring serach problem with provably good worst-case execution time and amortized number of I/Os, i.e. logarithmic I/O time in the input size. Furthermore, the absolute execution time doesn't dependent on input distribution, turning out to be a concurrent solution also for in-memory algorithms.

3.3 `Insert`$_{\texttt{SB}}$ Operation

By using our simplifications of the B-Tree definition, based on Cormen [46], the `Insert`$_{\texttt{SB}}$ operation turned out to be more simple and we show the essentials in the Algorithm 6.

Algorithm 2: `Insert`$_{\texttt{SB}}$ operation

```
Input: Key k, Value v
  // downwards recursive traversal of the B+-Tree
1 (new_separator, new_node) ← root.InsertUpDown_SB(k, v)
  // we've got an upcoming node at the root, so the must root split
2 if new_node ≠ null then begin
3     old_root ← root
4     root.Insert_PT(new_separator)
5     root.child[0] ← old_root
6     root.child[1] ← new_node
```

The Algorithm 37 shows the details of the recursive `InsertUpDown`$_{\texttt{SB}}$ function, fundamental for the `Insert`$_{\texttt{SB}}$'s work.

Algorithm 3: InsertUpDown$_{SB}$ auxiliary function of the Insert$_{SB}$ operation

Input: Root Node *root*, Key *k*, Value *v*

```
1  pos ← root.GetPosition_PT(k)
2  occupation ← root.GetOccupation()
3  if root.IsLeaf() then begin
4  |   leaf ← root
5  |   if leaf.IsFull() then begin
6  |   |   (left, right) ← leaf.Split_PT()
7  |   |   if pos >= occupation then begin
8  |   |   |   pos ← pos − occupation
9  |   |   |   target ← left
10 |   |   else begin
11 |   |   |   target ← right
12 |   |   target.Insert(pos, k, v)
13 |   |   new_node ← right
14 |   |   new_separator ← left.GetRightmost()
15 |   else begin
16 |   |   leaf.Insert(pos, k, v)
17 else if root.IsInternal() then begin
18 |   internal ← root
19 |   next_child ← internal[pos]
20 |   (new_separator, new_node) ← next_child.InsertUpDown_SB(k, v)
21 |   if new_child ≠ null then begin
22 |   |   if new_child ≠ null then begin
23 |   |   |   if internal.IsFull() then begin
24 |   |   |   |   (left, right) ← internal. Split_PT()
25 |   |   |   |   if pos > occupation then begin
26 |   |   |   |   |   pos ← pos − occupation
27 |   |   |   |   |   target ← left
28 |   |   |   |   else begin
29 |   |   |   |   |   target ← right
30 |   |   |   |   new_separtor ← right.ExtractLeftmostKey()
31 |   |   |   |   target.Insert(new_separator)
32 |   |   |   else begin
33 |   |   |   |   internal.Insert(new_separator)
34 |   |   |   |   right_k ← new_child.GetRightmostKey()
35 |   |   |   |   pos ← internal.GetPosition_PT(right_k)
36 |   |   |   |   internal[pos] ← new_child
37 return (new_separator, new_node)
```

3.4 Extended PTs

In the fundamentals we defined the basic PT and now we provide some extra definitions in order to describe some interesting properties of the PTs and to explain how to solve the predecessor problem via the $\texttt{GetPosition}_{\texttt{PT}}$ operation, despite of the PT's lossy compression. This extension allows for a better contextualization with the B^+-Tree.

The following extension of the PT allows for the reorganization of the B-Tree's layout during the dynamic operations $\texttt{Insert}_{\texttt{SB}}$ and $\texttt{Delete}_{\texttt{SB}}$. Since we don't focus on the $\texttt{Delete}_{\texttt{SB}}$ operation, only the $\texttt{Split}_{\texttt{PT}}$ and $\texttt{GetPosition}_{\texttt{PT}}$ operations are relevant for this thesis.

Definition 3.3. *(**Extended Patricia Trie**) [34] The Patricia Trie from the Definition 2.8 is extended with the following operations:*

- *The operation $\texttt{GetPosition}_{\texttt{PT}}(s)$ returns an integer j if exactly $j-1$ strings are lexicographically smaller than s in the string set $\mathcal{S}$, given any string $s \in \Sigma^*$.*
- *The operation $\texttt{Concatenate}_{\texttt{PT}}(PT_{\mathcal{S}_1},\ PT_{\mathcal{S}_2})$ joints two PTs $PT_{\mathcal{S}_1}$ and $PT_{\mathcal{S}_2}$, which represent two disjoint string sets S_1 and S_2, where $S_1 \leq S_2$.*
- *The operation $\texttt{Split}_{\texttt{PT}}(PT_{\mathcal{S}})$ splits the Patricia Trie $PT_{\mathcal{S}}$ into two PTs of nearly the same size in bytes.*

There are two interesting query operations, which can be answered by a trie: (i) the *membership* queries and (ii) the *lexicographical position* queries. The first has a binary answer, by simply determining whether a string is stored in the data structure or not. The second delivers a further detail, namely the query's lexicographical position among the elements stored there.

The lack of exact information about the keys stored in the PTs doesn't allow to distinguish the stored strings from the resting universe of strings. For example, it could be that the strings "*pin*" and "*pan*" can not be distinguished because one of them is not stored in the PT.

PTs can directly answer the first type of queries using a single I/O step. It comes out that PTs only store the strings' characters which are needed to distinguish any pair of different strings. Once the search algorithm stops at a leaf, it suffices to load the corresponding string and check whether the loaded string matches the query or not.

3.4.1 Serialization Methods

In order to efficiently answer the queries imposed to the String B-Tree internal nodes, explained in the previous subsections, the keys must be properly organized. Whenever dealing with memory-constrained systems or external memory algorithms, one such way would be to simply serialize these data structures so that they can be retrieved, reconstructed and used to be answer the queries. However reconstructing it from the scratch requires unnecessary computation steps that could make the String B-Tree not practical.

In order to make the String B-Tree ready for efficient use in practice, it is required to devise new operations and data structures to accelerate those calculations. One such way is to serialize/generate the internal node representations so that operations can be directly executed on the data without preprocessing steps (like restructuring it in main memory with pointers or unpacking it). The resulting data is a serialized data structure and the methods presented in this subsection allow for executing operations on those data without the need of main memory pointers.

To design our serialization methods, we base ourselves on the $\texttt{GetChild}_{\texttt{PT}}$ atomic operation, which takes a text symbol and a node as input and returns the root node of the corresponding

subtree reached by following the edge labeled by that input character. This atomic operation determines the execution time of the algorithms used to solve both the *membership problem* and the *predecessor problem*, namely the $\texttt{Search}_{\texttt{PT}}$ and $\texttt{GetPosition}_{\texttt{PT}}$ operations respectively.

Because we are also interested in solving the predecessor problem, in order to route search along the B-Tree, we can't afford probabilistic $\mathcal{O}(1)$ execution time for answering the $\texttt{GetChild}_{\texttt{PT}}$ operation. It happens that these running times can be reached mostly with randomization and this technique doesn't allow to establish a lexicographical order among the text characters. Thus, we relax this operation to be performed in $\mathcal{O}(\log|\Sigma|)$ execution time via binary search on the branching edges, while being able to answer the lexicographical position query for any input string.

Serializing a PT can be generally done using two main ideas, whenever operations must be efficiently done after serialization. The first one is to use one of the straightforward ways of traversing a multiway tree and to output its nodes sequentially, allowing for some operations like `Search`. A second approach would be to use multiple arrays in order to represent the different elements of the tree in separated arrays, allowing for a better domain compression and parallelization.

As for the first main idea, the most-popular traversal strategies are (i) pre-order, (ii) in-order, (iii) post-order and (iv) level-order. Pre-order firstly visits the root, than the smaller children, followed by the greater children recursively. In-order traversal relies on firstly visiting all the smaller children recursively first, than the root and afterwards the greater children in the same fashion done with the smaller children. Post order visits all the children first and than the root. Finally the level order considers a given node by visiting all its children at once in a breadth-first way and then repeating this procedure for all its children in lexicographical order.

For sake of cache-efficiency, we developed approaches to represent the tree structure that are based on the pre-order traversal. In this sense, we can make sure that for each tree node all $|\Sigma|$ edge labels, which are *possibly* going to be used for comparison before branching, reside on the cache-line. Designing properly the cache-line size according to the alphabet size allows for better performance prediction. Since the array is composed of text symbols and we used C chars during our initial implementations, we call it the *byte-based method*.

The second main idea is to store different tree parts, consisting of data with different data types, using different arrays. It allows for a space-optimal representation of a multiway tree. Furthermore, some computer architectures offer special instructions that can be used to operate efficiently on bits. Since the different parts of the tree are outsourced to multiple arrays which are used at the same time, we call it the *multiarray method*.

Byte-based Method

The main idea of this serialization approach is to conceptually traverse a trie and output the node labels in any given order. In our case we use a breadth-first order traversal to assure cache-efficiency. If not considering space-optimality, this serialization method is the best choice, because it represents the tree without pointers, while affording simple and fast atomic operations (e.g. the $\texttt{GetChild}_{\texttt{PT}}$ operation used by $\texttt{Search}_{\texttt{PT}}$ and $\texttt{GetPosition}_{\texttt{PT}}$).

Two additional symbols with the size of a text symbol are used to mark the begin of the internal nodes and the leaf nodes themselves. Given that the serialized tree is traversed by sequentially parsing subparts of the word representing the tree, one could argue that this approach is rather cache-efficient.

In practice, the tree node labels - the skip values and the string pointers - must be stored separately, since these have data types different from the text alphabet. Doing differently, in

order to keep all data in a single memory section, would cause extra space consumption for holding information to skip chunks of different data types, so that performance goals couldn't be fulfilled.

As skip values are needed during tree traversal, the cache-efficiency argument can be questionable, although this approach is much more cache-efficient than many other ones. From now on, we reserve the symbol `!` for marking the begin of an internal node and the symbol `&` for marking a leaf node. The actual labels can be fetched by performing `rank` operations on the whole word and using its result as the access index for using the labels array.

The following figure shows the tree resulting of inserting the string set $\mathcal{X} = \{$[0]`asdasd`, [1]`asdpsd`, [2]`bgfhg`, [3]`caaapp`, [4]`caaupp`, [5]`caaulp`$\}$.

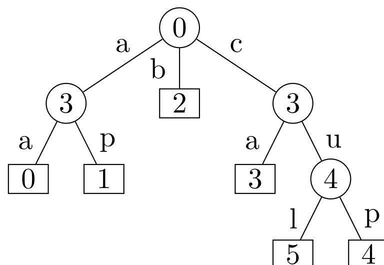


Figure 7: Patricia Trie example for serialization

The resulting pointerless representation of this PT is composed by the arrays depicted below. The array S_{BB} contains both the tree structure and the edge labels. The skip values of the internal nodes and the string pointers of the leaf nodes are stored in the arrays I_{BB} and L_{BB}, respectively.

$S_{BB} =$

!	a	b	c	!	a	p	&	&	&	!	a	u	&	!	l	p	&	&
0	1	2	3	4	5	6	7	8	9	10	11	12	14	14	15	16	17	18

$I_{BB} =$

0	3	3	4
0	1	2	3

$L_{BB} =$

0	1	2	3	4	5
0	1	2	3	4	5

Figure 8: Byte-based serialization representing the trie of Figure 7

As depicted above, the tree structure is designed to fulfill the demands for efficiently using the edge labels during tree traversal. The marks `!` and `&` allow for a simple loop-based tree traversal, so that the number of symbols to be skipped are calculated on-the-fly during search, by simply counting up a counter whenever a `!` symbols is read and decrementing it whenever a `&` symbol is read.

At the bottom line, the sequence of the edge labels reflect a preorder traversal of the tree, as the skip values' sequence reflects a level order traversal. The actual skip values are fetched by using the outcomes of applying the $\texttt{rank}_{!}(.)$ operation to access the I_{BB} array elements and the actual string pointers by applying the $\texttt{rank}_{\&}(.)$ operation to access the L_{BB} array elements.

In this way, we separate skip values, text symbols and string pointers for reducing space consumption and keep the code very simple to understand. Usually the number of bits used to represent a text symbol is much smaller than the number of bits to store a skip value or a string pointer, so that storing everything together would be not efficient, as mentioned before.

The order of the string pointers directly reflect the ordered string set, so that a bulk construction of the PT for full-text indexing only requires to copy of the SA entries into the array of string pointers.

Multiarray Method

This method follows ideas similar to the ones of the past subsection. However, here the focus is to separate the trie labels from the trie structure in order to group chunks of information of the same data type.

We start out by substituting the `&` and `@` marks by bits, which get stored in different arrays. On the one hand, it relatively worses the cache-efficiency of the algorithms, but on the other hand it enhances space usage by deriving a space-optimal data structure. The edge labels are stored at the same array positions used to store the corresponding structural information in the structure bit arrays, so that the multiarray operations can be performed in a simple way.

A node is identified by its subscript in the array of edge labels, E_{MA}, that is it is solely represented by its incoming edge, as the root has conceptually no corresponding part in the array E_{MA}. For any node, the rightmost child gets a bit set to one in the structure bit array S_{MA}, whereas the other nodes get their bits set to zero. Whether the nodes are leaves or not, it is defined by a further bit array T_{MA}, where a bit set to one indicates a leaf node and a bit set to zero indicates an internal node.

As in the byte-based approach, all the tree labels are stored separately from the skip values of the internal nodes, hold in the array I_{MA}, and the string pointers of the leaf nodes, hold in the array L_{MA}. The Figure 9 depicts those arrays and is also based on the PT from the Figure 7.

$E_{MA} =$

a	b	c	a	p	a	u	l	p
0	1	2	3	4	5	6	7	8

$S_{MA} =$

0	0	1	0	1	0	1	0	1
0	1	2	3	4	5	6	7	8

$T_{MA} =$

0	1	0	1	1	1	0	1	1
0	1	2	3	4	5	6	7	8

$I_{MA} =$

0	3	3	4
0	1	2	3

$L_{MA} =$

2	0	1	3	5	4
0	1	2	3	4	5

Figure 9: Multiarray serialization representing the trie of Figure 7

From the Figure 9 we see that the order of storing skip values and edge labels doesn't change, comparing to the byte-based approach. Still we can access three informations by querying a single array position.

How many array entries must be skipped, in order to reach the ith child node, is given by applying the $\texttt{rank}_0$ operation at the last bit set to 1 of the node in the structure bit array. For example, in the Figure 9, starting from the root, we follow the bit array from the index 0 up to the first structure bit set to 1, namely the entry indexed by 2, and calculate the $\texttt{rank}_0(2)$, giving us the total number of children. As next, we follow the next tree segments and accumulate their ranks in order to jump the subtrees, until the accumulated ranks equal zero.

The major drawback of the $\texttt{GetChild}_{\texttt{PT}}$ implementation using the arrays above is that the worst case depends on the subtree's size. If we are allowed to use extra memory space, we can precalculate the jump pointers in $\mathcal{O}(|\mathcal{S}|)$ internal time via depth-first search to get extra arrays for the subtree pointers, needed to spot the subarrays containing the subtree edges labels, and consequently the structure bits used for performing search. The following theorem summarizes this result.

Theorem 3.4. *The resulting Patricia Trie* PT *can be adapted in $\mathcal{O}(|\mathcal{S}_{\mathsf{PT}}|)$ internal time to support all operations in $\mathcal{O}(p \log |\Sigma|)$ internal time, while using at most $2|\mathcal{S}_{\mathsf{PT}}| \log |\mathcal{S}_{\mathsf{PT}}|$ extra bits for the internal node pointers.*

In the following, the array J^{lo}_{MA} denotes the extra array containing the jump pointers for accessing the i-th children's lower position in the array of edge labels E_{MA}. Since the arrays of leaf and structure bits S_{MA} and L_{MA} have the same semantics of the E_{MA} array, they can be also accessed using those jump pointers. Analogously, the array J^{hi}_{MA} contains the higher array positions, where the subtree's node/edge information ends. The Figures 28 and 29 show the state of the jump pointer arrays for the tree from the Figure 7.

$J^{lo}_{MA} =$

3	1	5	3	4	5	7	7	8
0	1	2	3	4	5	6	7	8

Figure 10: Higher jump pointers for multiarray serialization

$J^{lo}_{MA} =$

3	1	5	3	4	5	7	7	8
0	1	2	3	4	5	6	7	8

Figure 11: Lower jump pointers for multiarray serialization

Whenever traversing a node stored at the ith position, we query the lower and higher jump pointers at the slot indexed by i and get the range of outcoming edge labels to continue search. For example, given the tree from the Figure 7, we query the jump pointers at the position 0 to get the indices 3 and 4 that define the outgoing edges of the node stored at position 0. The node stored at position 0 is the internal node labeled with 3, found by traversing the edge labeled with a from the root. The labels of the outgoing edges stored in the range between 3 and 4 are a and p, which simultaneously define the leaf nodes labeled with the string pointers 0 and 1.

As previously mentioned, we follow nearly the same conventions for organizing the data along the multiple arrays, except for the order of the string pointers is different. We use the level-order induced by a breadth-first search, rather than the ordering induced by a depth-first search (consequently the lexicographical order between the strings).

More exactly, the leaf marks of the byte-based approach are directly embedded in the tree structure representation, so that they uniquely reflect the lexicographical string order of the leaves, because we apply a preorder traversal to find them and can spot them uniquely within the tree word S_{BB}. In the multiarray method, the positions of the leaf bits correspond to positions of the nodes which simultaneously represent their incoming edge labels and the nodes themselves. Thus, ranking the leaf bits during tree traversal to find the lexicographical order of the string is an ambiguous operation. A one-to-one correspondance between leaf bits and string pointer indices can be found only if the order of the string pointers directly reflect the level-order strategy. Doing so simplifies the implementation of the $\texttt{GetPosition}_{\texttt{PT}}$ operation, like finding the leftmost descendent leaf of a given internal node.

Complexity Summary

In the following, Σ denotes the text alphabet, as usual, Π denotes the node labels alphabet (skip values alphabet) and Γ denotes the string pointers alphabet. We let $\Sigma = \Pi = \Gamma = \mathcal{A}$ in

order to simplify notation, leaving aside exactness what is not relevant for this comparison. The Fast Multiarray serialization corresponds to the multiarray method with extra jump pointers, as the Slow Multiarray serialization corresponds to the raw multiarray method without them.

Serialization	**Index Space** (bits)	**Search** (comparisons)
Byte-based	$2(\lvert\mathcal{S}\rvert \log \lvert\Sigma\rvert) + 3\lvert\mathcal{S}\rvert \log \lvert\mathcal{A}\rvert$	$\mathcal{O}(\lvert\mathcal{S}\rvert)$
Fast Multiarray	$2\lvert\mathcal{S}\rvert + 3\lvert\mathcal{S}\rvert \log \lvert\mathcal{A}\rvert + 2\lvert\mathcal{S}\rvert \log \lvert\mathcal{S}\rvert$	$\mathcal{O}(p \log_2 \lvert\Sigma\rvert)$
Slow Multiarray	$2\lvert\mathcal{S}\rvert + 3\lvert\mathcal{S}\rvert \log \lvert\mathcal{A}\rvert$	$\mathcal{O}(\lvert\mathcal{S}\rvert)$

Table 3: Worst-case summary table for the Patricia Trie representations

Clearly bounding the execution time of the `Search` operation on the pattern's size p is better than bounding it on the size of the Patricia Trie $|\mathcal{S}|$.

3.4.2 `GetPosition`$_{\texttt{PT}}$ Operation

We solve the more generalized type of query, which asks for the lexicographical position of a string in a set. The solution is described as the `GetPosition`$_{\texttt{PT}}$ operation.

Two steps are needed in order to achieve exactness with high performance in external memory models. The first step only uses the branch characters of the blind trie to reach one of the leaves sharing the maximum LCP with the query. A second step performs a (potentially full) string comparison to find the correct position among those leaves and ensure answer exactness. The first step has no external work, as the second does it with minimal number of I/O operations.

The string set $\mathcal{S}$ is characterized by the lexicographical order $\leq_L$, as the PT has the LCP information encoded as the *skip values* labeling internal nodes. The relationship between both can be precised with the following fact:

Fact 3.5. *[34] For any strings X_1, X_2 and Y, such that either $X_1 \leq_L X_2 \leq_L Y$ or $Y \leq_L X_2 \leq_L X_1$: $lcp(X_1, Y) \leq lcp(X_2, Y)$.*

Informally speaking, by considering any two neighboring strings they can have a greater LCP value than the LCP value between one of them and a string which is lexicographically farther than second party. It means that strings which are lexicographically "closer" in the ordered string set have a deeper corresponding lowest common ancestor (LCA) in the induced trie. This is a useful fact to be exploited when using preprocessed range minimum queries (RMQs), in order to reconstruct the trie structure using the LCP information in $\mathcal{O}(|\mathcal{S}|)$ internal running time [20].

Blind Search

We perform a *downward traversal* from the root to potentially find a leaf. From the root down, P's characters are subsequentially compared with the branching characters of the edges to match some of P's characters. At some point, we either (1) reach a leaf or we (2) can not branch anymore:

1. If the algorithm stops at a leaf l, then we had no errors when using the compressed trie. That means the string found *must* provide us the maximum LCP value, because if any other, then we would either branch differently or get at least one mismatch. The first case is not possible, because the edge labels are unique for each node and the second case is not possible because we would have stopped at an internal node, if so.

2. If the hit node u is an internal node, from Property 3 of Definition 2.8 all descendent leaves reached from u share a common prefix. This means that all descendent leaves are equivalent for calculating the maximum LCP value, so that choosing any leaf l' satisfies the condition

$$lcp(\text{string}(l'),\, P) = \text{max_LCP}(P,\, \mathcal{S}).$$

Below we show the algorithm for performing the `BlindSearch` operation (Algorithm 8), meanwhile hidding some implementation details, like defining branches which are smaller than all or greater than all edge labels. Those details don't change the core mechanics of the problem's solution and make the text much more readable.

Algorithm 4: $\texttt{BlindSearch}_{\texttt{PT}}$ operation

```
Input: Query P, Node node
1 skip_value ← node.GetSkipValue_PT()
2 symbol ← P[skip_value]
3 rank ← node.GetRank_PT(symbol)
4 if node.IsMismatch(P, rank) or node.IsPrefix(P, rank) then begin
5 |   return node
6 else begin
7 |   subtree ← node.GetChild_PT(rank)
8 |   return subtree.BlindSearch_PT(pattern)
```

In case the resulting node has a *skip value* smaller than the actual maximum LCP value, the following verification step corrects it by properly comparing the strings from the leaf found during blind search.

Verification Step

This step determines the exact P's position in $\mathcal{S}$. First, the LCP value between the string belonging to leaf from $\texttt{BlindSearch}_{\texttt{PT}}$ and the pattern is computed, i.e. $lcp = \texttt{lcp}(\text{string}(l), P)$. Secondly, we derive the mismatching characters $c = P[lcp + 1]$ and $c' = \text{string}(l)[lcp + 1]$ that are well-known (see Fact 3.4 from [34]).

In more details, we are given the hit node resulting from the $\texttt{BlindSearch}_{\texttt{PT}}$ and its label L (skip value), which is less or equal the actual maximum LCP value, but not greater (see Lemma 3.5 from [34]). From that node down we find a leaf which gives the max_lcp $\geq L$. Note that we must compare only the resting $lcp - L$ characters, what results in less disk blocks to be loaded, namely only $\frac{lcp-L}{B} + 1$ amortized I/O operations are executed for string comparison. This step can significantly save ressources, if the texts are big and this operation is very often repeated.

Given the hit node u resulting from the $\texttt{BlindSearch}_{\texttt{PT}}$, if $skip_value[u] = lcp$, we use the branching edges to find out what subtree contains the correct separator. This can be done via binary search in $\mathcal{O}(\log|\Sigma|)$ time, if the labels are stored sorted. Otherwise, if $skip_value[u] > lcp$ we are lucky, because all descendents share the same prefix, thus all of them give the same information and we just compare the mismatching characters to follow either all the way to the right or all the way to the left, getting the right leaf with no further I/O operations.

Putting blind search and verification step together

Combining the results from the procedures above we find the following algorithm for finding the exact position of a pattern using the PT. The function $\texttt{FindProperLeaf}_{\textsf{PT}}$ handles all special cases explicitly handled in the C++ code, which tend to be simple but specific to the serialization methods (e.g. situations, where the pattern's character is lexicographically smaller than all branching characters).

We perform two $\texttt{BlindSearch}_{\textsf{PT}}$ operations in $\mathcal{O}(|\mathcal{S}|)$ internal time and one I/O step. More exactly, in order to decide upon what leaf must be taken to determin P's position, we need to go down the tree more one time costing more $\mathcal{O}(|\mathcal{S}|)$ internal time. Overall, we need $\mathcal{O}(|\mathcal{S}|)$ internal time plus $\mathcal{O}(1)$ external time to find the P's position within the range determined by the String B-Tree's internal node.

The Algorithm 22 shows procedure for finding the position of any string in a set of strings using the Extended Patricia trie. We expose the most important details of the procedure's mechanics with an abstract algorithm, which is exact enough to be used for precise analysis of our implementations. Some implementation details are hidden, for being very specific to the serialization approaches (e.g. how to handle an edge containing the sentinel character '$') and these don't influence the complexity analysis.

Further, we only approach the operations needed to perform the $\texttt{Insert}_{\textsf{SB}}$ operation, namely the $\texttt{Split}_{\textsf{PT}}$ and $\texttt{Insert}_{\textsf{PT}}$ operations. The $\texttt{Delete}_{\textsf{SB}}$ operation is not relevant for experimental evaluation and is not approached in this thesis.

3.4.3 $\texttt{Insert}_{\textsf{PT}}$ Operation

The $\texttt{Insert}_{\textsf{PT}}$ operation performs the verified search to get a hit node h and determine a position induced by that node, where the structures representing the new string s must be put. That position depends on the LCP value between the input string and the string represented by the hit node (or any of its descendents, in case it is an internal node). For that, we compare the mismatching character of the input string with the $\mathcal{O}(|\Sigma|)$ branching characters of the found node, in case LCP(s, string(l) $\geq$ $skip_value(h)$) for any descending leaf l, and put the new leaf at the found position. Otherwise, we put a new internal node with extra edges between h and $parent(h)$.

Performing the $\texttt{Insert}_{\textsf{PT}}$ operation on pointerless representations of a PT requires shifting blocks of memory, in order to rewrite the subtrees affected by the tree transformations. We need at least one shift to put the new branching character and one shift to put the new string's pointer. Further data must be also put, depending on the serialization strategy and usually, the more the trie is compressed, more memory shifts are required.

Leaving string pointers and skip values aside, we can observe that a trie represented using the byte-based method shifts memory only two times and puts two blocks of data inbetween. Further, those shifts are dependent on each other and can't be parallelized. It happens that the node/leaf marks are stored just before the corresponding node edge labels. Although consuming extra symbols with size of an alphabet symbol, it offers more flexibility for determining the cut positions. The multiarray method completely separates the tree structure from the labels and that means we can transform each array in parallel, once we determined the cut positions. In fact, we move more memory, but on the other hand we can do it in parallel and the algorithms are somewhat more efficient.

The operations $\texttt{PutBranchForLeaf}_{\textsf{PT}}$, $\texttt{PutLeafForInternal}_{\textsf{PT}}$ and $\texttt{PutBranchForInternal}_{\textsf{PT}}$ cover all possible tree transformations, so that the $\texttt{Insert}_{\textsf{PT}}$ operation can be assembled using these basic ingredients. The Figures 13, 12 and 14 depict these transformations, where the

Algorithm 5: GetPosition$_{PT}$ operation

```
Input: Pattern P

   // find a max_LCP leaf via blind search
 1 hit_node ← BlindSearch_PT(P)
 2 if hit_node.IsLeaf() then begin
 3   hit_pointer ← GetStringPointer_PT(hit_node)
 4 else begin
 5   hit_pointer ← GetStringPointer_PT(GetLeftmostLeaf_PT(hit_node))
   // perform the verification step via a second LCP-constrained BlindSearch_PT
 6 lcp ←CalcLCP(hit_pointer, P)
 7 c ← P[lcp + 1]
 8 c' ← GetString(hit_node)[lcp + 1]
 9 hit_node ← BlindSearch_PT(P, lcp)
   // find P's position by finding its correct leaf
10 if hit_node.IsLeaf() then begin
11   return CountPreceedingLeaves_PT(hit_node)
12 else begin
13   skip_value ← GetSkipValue_PT(hit_node)
14   if lcp < skip_value then begin
15     if c ≤_L c' then begin
16       return CountPreceedingLeaves_PT(GetLeftmostLeaf_PT(hit_node))
17     else begin
18       return CountPreceedingLeaves_PT(GetRightmostLeaf_PT(hit_node))
19   else if lcp = skip_value then begin
20     rank ← hit_node.GetChildRank_PT(c)
21     subtree ← FindProperLeaf_PT(hit_node.GetChild_PT(rank))
22     return CountPreceedingLeaves_PT(subtree)
```

dashed parts denote the tree parts found during search and the red structures are the new tree parts.

Algorithm 6: `Insert`$_{\texttt{PT}}$ operation

```
Input: Key P

   // find the node where to put the leaf
1  hit_node ← BlindSearch_PT(P)
2  if hit_node.IsLeaf() then begin
3  |  hit_pointer ← GetStringPointer_PT(hit_node)
4  else begin
5  |  hit_pointer ← GetStringPointer_PT(GetLeftmostLeaf_PT(hit_node))
6  lcp ← CalcLCP(P, hit_pointer)
7  c ← P[lcp + 1]
8  c' ← GetString(hit_pointer)[lcp + 1]
9  verified_node ← BlindSearch_PT(P, lcp)
   // rewrite the tree by putting the a leaf in the trie
10 if verified_node.IsLeaf() then begin
11 |  PutBranchForLeaf_PT(verified_node, c, c')
12 else begin
13 |  skip_value ← GetSkipValue_PT(verified_node)
   |  // branch at hit node
14 |  if lcp = skip_value then begin
15 |  |  PutLeafForInternal_PT(verified_node, c, c')
   |  // branch between hit node and its parent
16 |  else if lcp < skip_value then begin
17 |  |  PutBranchForInternal_PT(verified_node, c, c')
```

After describing the pointerless representations of the PT and explaining the tree transformations belonging to the `Insert`$_{\texttt{PT}}$ operation, we briefly give three examples of how the tree transformations change the data structure. For that, we recall the PT of the Figure 7, which initially holds the string set $\mathcal{X}$ = {[0]`asdasd`, [1]`asdpsd`, [2]`bgfhg`, [3]`caaapp`, [4]`caaupp`, [5]`caaulp`}.

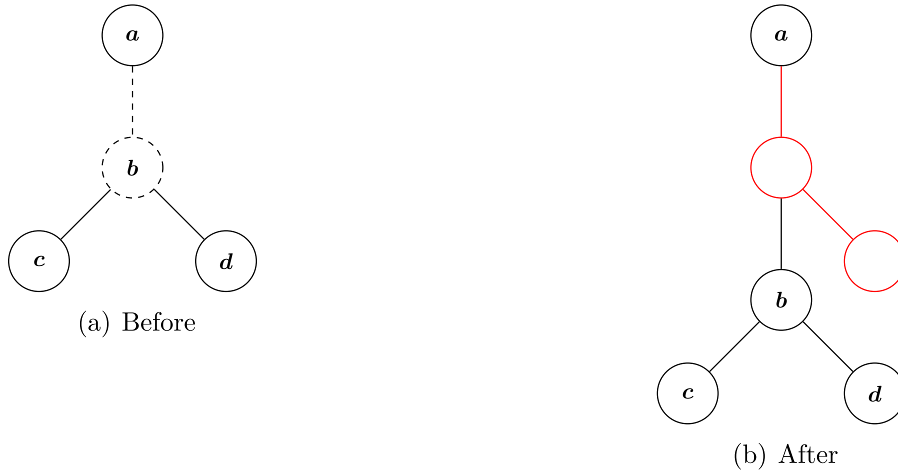


Figure 12: Trie transformation: $\texttt{PutBranchForInternal}_{\text{PT}}$

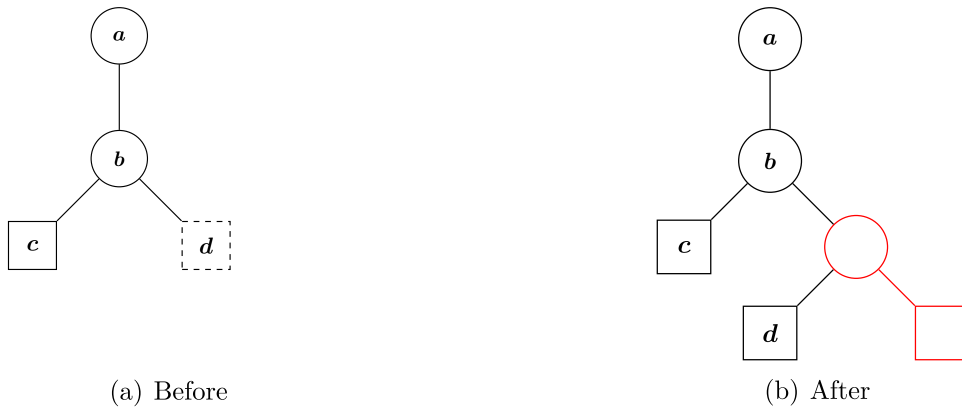
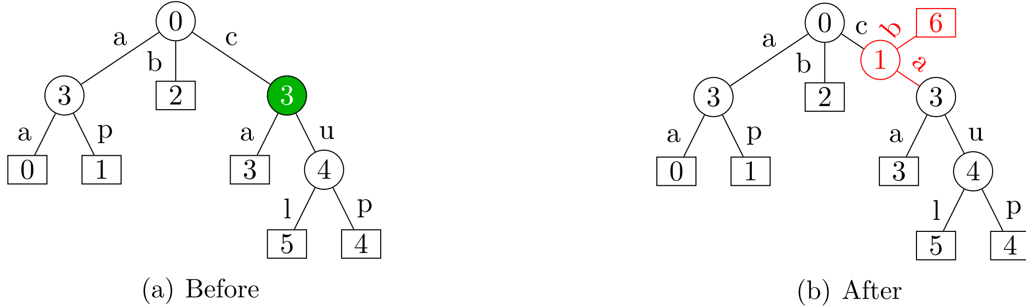


Figure 13: Trie transformation: $\texttt{PutBranchForLeaf}_{\text{PT}}$

Figure 14: Trie transformation: $\texttt{PutLeafForInternal}_{\text{PT}}$

Example of trie transformation: `PutBranchForInternal`$_{\mathrm{PT}}$

The key "`cbt`" is inserted. After performing search, we hit the node h colored with green below. Since LCP(s, string(l)) $<$ *skip_value*(h) for all descendent leaves l, we add a new internal node with further edges between the hit node h and `parent`(h):



(a) Before (b) After

Figure 15: Example of trie transformation: `PutBranchForInternal`$_{\mathrm{PT}}$ operation

Changes for the multiarray serialization:

E_{MA} =	a	b	c	a	p	a	u	l	p
	0	1	2	3	4	5	6	7	8

S_{MA} =	0	0	1	0	1	0	1	0	1
	0	1	2	3	4	5	6	7	8

T_{MA} =	0	1	0	1	1	1	0	1	1
	0	1	2	3	4	5	6	7	8

I_{MA} =	0	3	3	4
	0	1	2	3

L_{MA} =	2	0	1	3	5	4
	0	1	2	3	4	5

Figure 16: Before `PutBranchForInternal`$_{\mathrm{PT}}$

E_{MA} =	a	b	c	a	p	a	b	a	u	l	p
	0	1	2	3	4	5	6	7	8	9	10

S_{MA} =	0	0	1	0	1	0	1	0	1	0	1
	0	1	2	3	4	5	6	7	8	9	10

T_{MA} =	0	1	0	1	1	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8	9	10

I_{MA} =	0	3	1	3	4
	0	1	2	3	4

L_{MA} =	2	0	1	6	3	5	4
	0	1	2	3	4	5	6

Figure 17: After `PutBranchForInternal`$_{\mathrm{PT}}$

Changes for the byte-based serialization:

S_{BB} =	!	a	b	c	!	...	l	p	&	&
	0	1	2	3	4		15	16	17	18

I_{BB} =	0	3	3	4
	0	1	2	3

L_{BB} =	0	1	2	3	4	5
	0	1	2	3	4	5

Figure 18: Before `PutBranchForInternal`$_{\mathrm{PT}}$ [8]

S_{BB} =	!	...	a	u	!	a	b	...	&	&
	0	...	11	12	13	14	15	...	20	21

I_{BB} =	0	3	1	3	4
	0	1	2	3	4

L_{BB} =	0	1	2	3	5	4	6
	0	1	2	3	4	5	6

Figure 19: After `PutBranchForInternal`$_{\mathrm{PT}}$

Example of trie transformation: `PutBranchForLeaf`$_{PT}$

The key s = "`asdpsdw`" is to be inserted. After performing search, we hit the node h colored with green below. Since string(h) is a prefix of s, we substitute this leaf by two leaves, one for old and one for the new string:

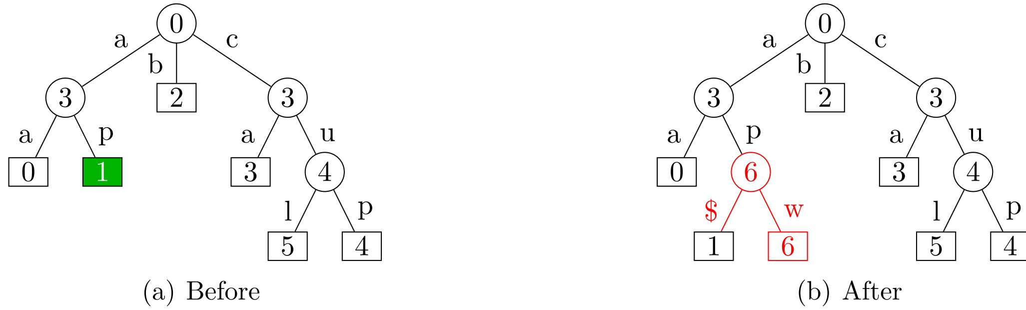


Figure 20: Example of trie transformation: `PutBranchForLeaf`$_{PT}$ operation

Modifications for the multiarray serialization:

E_{MA} =

a	b	c	a	p	a	u	l	p
0	1	2	3	4	5	6	7	8

S_{MA} =

0	0	1	0	1	0	1	0	1
0	1	2	3	4	5	6	7	8

T_{MA} =

0	1	0	1	1	1	0	1	1
0	1	2	3	4	5	6	7	8

I_{MA} =

0	3	3	4
0	1	2	3

L_{MA} =

2	0	1	3	5	4
0	1	2	3	4	5

Figure 21: Before `PutBranchForLeaf`$_{PT}$

E_{MA} =

a	b	c	a	p	$	w	a	u	l	p
0	1	2	3	4	5	6	7	8	9	10

S_{MA} =

0	0	1	0	1	0	1	0	1	0	1
0	1	2	3	4	5	6	7	8	9	10

T_{MA} =

0	1	0	1	1	0	1	1	0	1	1
0	1	2	3	4	5	6	7	8	9	10

I_{MA} =

0	3	6	3	4
0	1	2	3	4

L_{MA} =

2	0	1	6	3	5	4
0	1	2	3	4	5	6

Figure 22: After `PutBranchForLeaf`$_{PT}$

Modifications for the byte-based serialization:

S_{BB} =

!	a	b	c	!	...	l	p	&	&
0	1	2	3	4		15	16	17	18

I_{BB} =

0	3	3	4
0	1	2	3

L_{BB} =

0	1	2	3	4	5
0	1	2	3	4	5

Figure 23: Before `PutBranchForLeaf`$_{PT}$ [8]

S_{BB} =

!	...	&	!	$	w	&	&	&	...
0	...	7	8	9	10	11	12	13	...

I_{BB} =

0	3	6	3	4
0	1	2	3	4

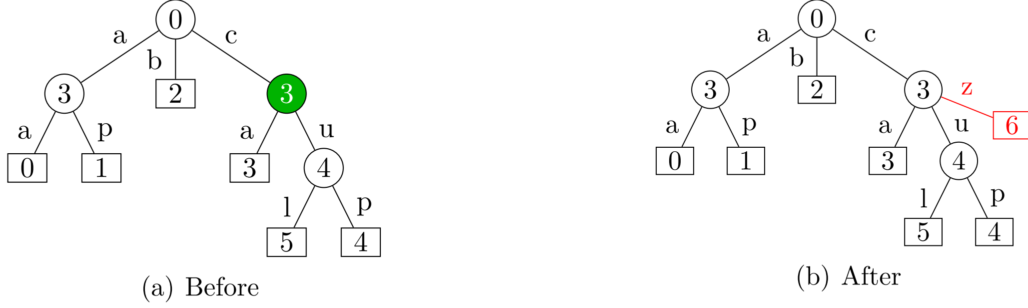
L_{BB} =

0	1	6	2	3	5	4
0	1	2	3	4	5	6

Figure 24: After `PutBranchForLeaf`$_{PT}$

Example of trie transformation: `PutLeafForInternal`$_{\text{PT}}$

The key s = "`caaz`" is to be inserted. After performing search, we hit the node h colored with green below. Since LCP(s, string(l)) = $skip_value(h)$ for all descendent leaves l, we add a new leaf branching from the hit node:



(a) Before (b) After

Figure 25: Example of trie transformation: `PutLeafForInternal`$_{\text{PT}}$ operation

Modifications for the multiarray serialization:

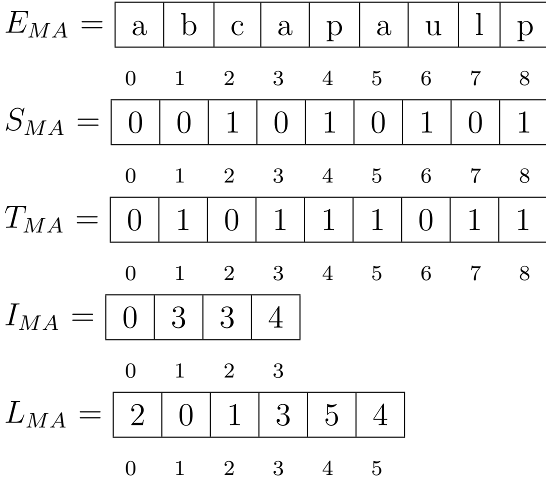


Figure 26: Before `PutLeafForInternal`$_{\text{PT}}$

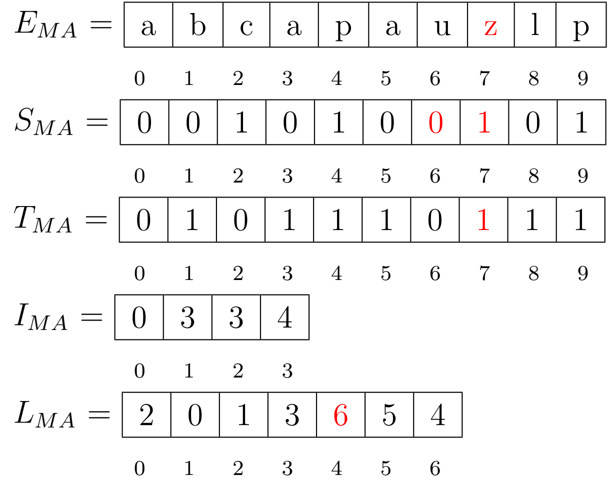


Figure 27: After `PutLeafForInternal`$_{\text{PT}}$

Modifications for the byte-based serialization:

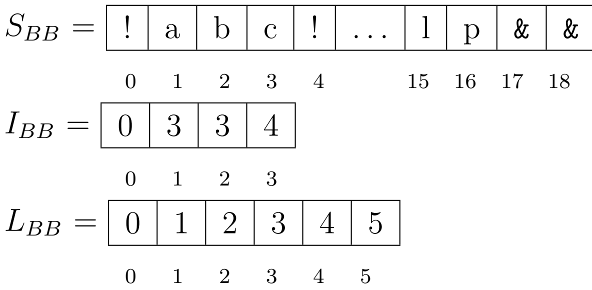


Figure 28: Before `PutLeafForInternal`$_{\text{PT}}$ [8]

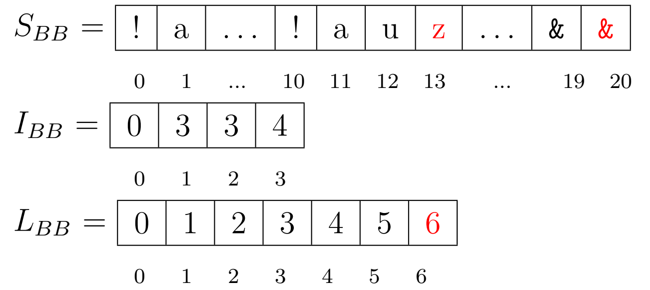


Figure 29: After `PutLeafForInternal`$_{\text{PT}}$

To close our discussion about the `Insert`$_{\texttt{PT}}$ operation, we show the PT size growths in bits, whenever the different transformations are applied. Knowing those values can be useful for implementing optimized allocators. To get realistic values, we consider text symbols of one byte, skip values of 4 bytes and string pointers of 8 bytes. NA denotes the naive approach, where each PT node holds 8 byte pointers for its children nodes.

Transformation	MA	BB	NA
PutBranchForInternal$_{\texttt{PT}}$	116 bits	128 bits	192 bits
PutBranchForLeaf$_{\texttt{PT}}$	116 bits	128 bits	192 bits
PutLeafForInternal$_{\texttt{PT}}$	107 bits	112 bits	176 bits

Table 4: Absolute index growth per insertion transformation

Putting the numbers on highlight, we can realize, for example, that an index built using the MA approach can save nearly 150 MB whenever the input size equals 100.000.000. In the section 5 we analyze how those effects happen in practice, as the table below show the worst-case values.

Input Size	Space Usage Savings
100.000.000	143.05 MB ($\approx$ 0.1 GB)
1.000.000.000	1,430.51 MB ($\approx$ 1.5 GB)
100.000.000.000	143,051.14 MB ($\approx$ 100 GB)

Table 5: Worst-case space usage savings of the MA serialization against the BB serialization

3.4.4 Split$_{\texttt{PT}}$ Operation

Defining the `Split`$_{\texttt{PT}}$ operation depends on how we define the tree weight and typically, graph partitioning problems fall under the NP-hard problems, even for special graph classes such as trees. We define the `Split`$_{\texttt{PT}}$ operation referent to a fixed representation and based on the requirements of the `Insert`$_{\texttt{SB}}$ operation: the trie must be splitted in such a way to induce a balanced B-Tree with less fragmentation. To reach it, we use the trie's space consumption in bytes as weight and define the operation as follows.

Definition 3.6. ***(Splitted Trie)*** *Given an encoding scheme $\mathcal{P}$ for representing multiway trees and an arbitrary trie t, the result of splitting $\mathcal{P}(t)$ is a pair of tries $\mathcal{P}(t_1)$ and $\mathcal{P}(t_2)$ with $w(\mathcal{P}(t_1)) \approx w(\mathcal{P}(t_2))$, where the weight function $w(.)$ associates a trie representation $\mathcal{P}(t)$ to a natural number $n \in \mathcal{N}_+$, the trie's weight. The weight is defined as the number of bytes needed to represent the trie using the encoding scheme $\mathcal{P}$.*

Since the number of nodes of a PT is not difficult to estimate even under arbitrary alphabets, we just adopt two simple approaches to split a trie. The first scheme considers the root and use the root's degree to generate two subtrees. The first half of the edges is traversed towards the leaves to generate a first trie, and the second edge half is traversed to generate a second trie in the same fashion. We call this approach as `Fast-Split`, since it takes only $\mathcal{O}(|\mathcal{S}|)$ internal time to finish and makes no random disk access. The main disadvantage is that the algorithm is recursive and some special cases must be handled, like when the root has a single child. Besides handling specific implementation issues, this method doesn't exploit the ease of estimating a

PT's size based on its number of strings. For example, a root with only two children, where one subtree has a single descendent leaf and the other has $|\mathcal{S}| - 1$ descendent leaves, would generate very bad results.

A second strategy is to simply create new tries by inserting half part of the strings into one trie and the second half into a second trie. Asymptotically it takes $\mathcal{O}(|\mathcal{S}|^2)$ internal time to finish and makes $|PT_{\mathcal{S}}|$ disk accesses for completely generate the tries from the sratch. The main advantage regards the procedure's correctness, which only depends on the correctness of the $\texttt{Insert}_{\texttt{PT}}$ operation. The quadratic internal work complexity can be in practice not a serious issue, because both tries can be generated independently in parallel. Furthermore, the $\texttt{Split}_{\texttt{PT}}$ operation is used neither during the bulk construction nor during searching, being the less used case, what prioritizes the simplicity against performance.

3.5 Bulk Construction

Bulk constructing the String B-Tree has several advantages against the iterative construction process. Firstly, it is parallelizable at both fine- and coarse-grained levels. Secondly, the algorithms involved are very reliable and are not bounded on the $\texttt{Split}_{\texttt{PT}}$ operation used by the $\texttt{Insert}_{\texttt{SB}}$ operation. Those procedures are heuristics and their outcomes are not efficient in respect to the resulting tree layout.

We are interested in handling two specific construction problems: (i) the bulk construction out of an array of string pointers sorted in lexicographical order, accompanied by the LCP array, and (ii) the bulk construction out of the SA, accompanied by the LCP array. The problem essence is the same in both cases, whereas (i) is the more general case, where the strings are not necessarily all the suffixes of a single string. In our examples we solve the more general problem.

Regarding parallelization at fine-grained level, multiple threads performing bulk construction using disjointed subparts of the SA and LCP arrays can be simultaneously triggered using the *Fork-Join Model* [6], operating level-by-level in the B-Tree structure. Coarse-grained parallelization can also be exploited, by distributing the B-Tree nodes over diverse computer nodes, so that the $\texttt{Search}_{\texttt{SB}}$ operation works on a tree of computers and supports subsequential search requests without query acceptance delay, increasing the system's throughput.

In the following, we describe the algorithm for bulk constructing a single PT using the SA and LCP arrays as input in the Subchapter 3.5.1. Then, in the Subchapter 3.5.2 the level-by-level recursive construction algorithm of the String B-Tree is explained and briefly analyzed.

3.5.1 Patricia Trie

The basic idea of the PT construction algorithm is the observation that the labels of the internal nodes are nested in the subranges of the LCP array, as depicted in the Figure 31. The edge labels can be extracted from the original text using the LCP array values, possibly combined with SA entries. Reading out the mismatching characters and the LCP array is enough to completely construct the PTs without performing any further I/O.

The trie used in the examples is depicted in Figure 30 and it holds the strings depicted in the Figure 31(a). Its structure is complex enough to show the mechanics of our algorithm.

The source data is plotted in the Figure 31(a), as the Figure 31(b) shows the subtree parts corresponding to the subranges. The input strings S_i are lexicographically sorted and the column LCP contains the LCP values, where the line i holds LCP(i, $i-1$), except for the first line which holds a dummy 0 value. The column Pair contains the mismatching characters

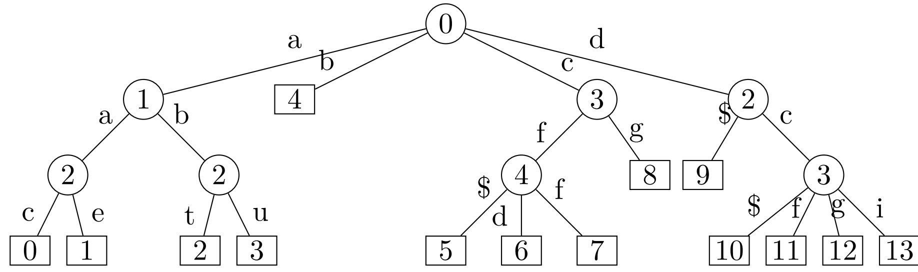
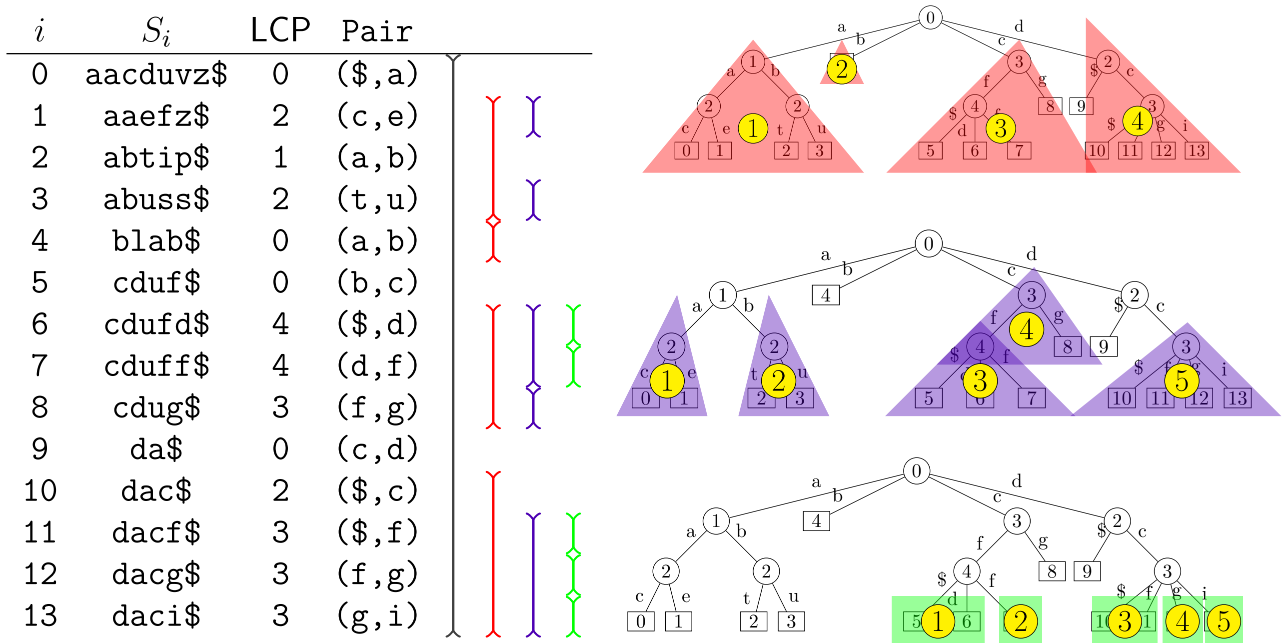


Figure 30: Patricia Trie example for bulk construction

occuring between two subsequent strings, i.e. the character pair (S_{i-1}[LCP[i] + 1], S_i[LCP[i] + 1]), except for the row with $i = 0$, which holds a dummy pair (\$, S_0[0]).

All the dummy values are put, so that the indices of the SA, LCP array, input strings and the pairs of mismatching characters can be used in an uniform way. The subranges are represented in the Figure 31(a) using two natural numbers, lo and hi, and we depict them with segment lines enclosed by two arrows at the extremes pointing to the line middle. The subranges are recursively unfolded until a leaf is reached. The gray segments depict subranges resulting from the first level of recursion, namely the single subrange ranging from 0 to 13. The red segments depict the subranges of the second level of recursion, the purple segments depict the ones resulting from the third level, as the green ones depict the fourth level subranges.

In the Figure 31(b) the subtree parts are numbered in ascending order to correspond the subrange segments of Figure 31(a) from the uppermost subrange segment to the lowermost subrange segment.



i	S_i	LCP	Pair
0	aacduvz$	0	($,a)
1	aaefz$	2	(c,e)
2	abtip$	1	(a,b)
3	abuss$	2	(t,u)
4	blab$	0	(a,b)
5	cduf$	0	(b,c)
6	cdufd$	4	($,d)
7	cduff$	4	(d,f)
8	cdug$	3	(f,g)
9	da$	0	(c,d)
10	dac$	2	($,c)
11	dacf$	3	($,f)
12	dacg$	3	(f,g)
13	daci$	3	(g,i)

(a) Source data with subranges on rightmost side

(b) Subtree parts corresponding to the subranges

Figure 31: Correspondance between the SA and LCP arrays, and the Patricia Trie

The leaves are represesented by segments for which lo equals hi. Otherwise, if lo not equals hi

we have a subtree part, where the nodes are disposed around their parents. One can visialize the structure better thinking in terms of the in-range smallest LCP value ($\texttt{min}_{lo..hi}$) and observing the array entries surrounding it. For example, in the range 0..13 the smallest LCP value equals 0. So, the arrays are conceptually traversed from 0 to 13 and a leaf is found whenever LCP[i+1] = LCP[i] = $\texttt{min}_{0..13}$ = 0, what is valid for $i = 4$. Otherwise, given that LCP[j] = $\texttt{min}_{0..13}$, a subtree part is found counting up from the smaller position higher than j up to the greater position smaller than the next occurance of $\texttt{min}_{0..13}$, what is valid for $i \in \{6, \ldots, 8\}$.

Note that even though we mentioned that the subranges are traversed from the begin to the end, it doesn't happen in practice, so that the algorithm runs in linear internal time. It suffices to see that any leaf or subtree root is visited only once. At a next step, only the array entries correspodning to the actual subtree parts are unfolded. This unfold step is done via range minimum queries (RMQs), so that the subtree root nodes are touched only once via those queries, the leaves break the recursion and the subranges are recursively unfolded.

We show the exact steps to bulk constrct the PT by showing the Algorithm 31. The variables have a correspondance with the notation introduced in the Subchapter 3.4.1 and we use the MA method to represent the unfolded the PT. We let *edges* denote E_{MA}, *structure* denote S_{MA}, *leaves* denote T_{MA}, *skips* denote I_{MA} and *pointers* denote L_{MA}.

Before exposing the actual PT bulk construction algorithm, we brifly show the routine $\texttt{GetRanges}_{\texttt{PT}}$ used to unfold the subranges. Its running time is $\mathcal{O}(|\mathcal{S}|)$, but an amortized argumentation proofs that the Algorithm 31 runs in $\mathcal{O}(|\mathcal{S}|)$ internal time whenever using this auxiliary function, since we operate on proper subranges.

Algorithm 7: $\texttt{GetRanges}_{\texttt{PT}}$ auxiliary function of the $\texttt{BulkConstruction}_{\texttt{PT}}$ operation

Input: LCP Array *lcp_array*, Lower Index *lo*, Higher Index *hi*

```
1  (lo, min) ← GetMinInRange(lo, hi)
   // retrieve the indices of all occurances of the minimum LCP value min
2  while lo < hi do
       // retrieve the first occurance of the minimum LCP in the range {lo,...,hi} in O(1)
3      (lo, value) ← GetMinInRange(lo, hi)
4      if value == min then begin
5          lowers.append(lo) ; lo ← lo + 1
6      else begin
7          break
   // calculate the higher indices from the sequence of lower indices
8  for i in 1, ..., lowers.size() do
9      lo ← lowers[i]
10     if lcp_array[lo + 1] == min then begin
11         hi ← lo
12         ranges.append(lo, hi)
13     else begin
14         lo ← lo + 1 ; hi ← indices[i + 1] − 1
15         ranges.append(lo, hi)
```

Algorithm 8: BulkConstruction$_{\mathtt{PT}}$ operation

```
Input: Suffix Array sa, LCP Array lcp_array, Lower Bound lo, Higher Bound hi, LCP lcp

   // load pairs of mismatching characters using the suffix and LCP arrays
1  pairs ← GetMismatchPairs(lcp, sa)
   // initialize auxiliary data structures
2  for i in 1, ..., sa.size() do
3  |  visited[i] ← 0
4  |  leaf_mark[i] ← lcp_array[i + 1] ≤ lcp_array[i] ? 1 : 0
   // serialize a leaf node
5  if lo = hi then begin
6  |  if visited[lo] = 0 then begin
7  |  |  skips.append(lcp_array[lo])
8  |  |  edges.append(pairs[lo].first) ; edges.append(pairs[lo].second)
9  |  |  structure.append(0) ; structure.append(1)
10 |  |  leaves.append(1)
11 |  |  pointers.append(sa[lo])
   // generate a list of proper subranges
12 ranges ← GetRanges_PT(lo, hi)
   // serialize an internal node
13 if lo ≠ hi then begin
14 |  skips.append(lcp)
15 for i in lo, ..., hi do
16 |  if lcp_array[i] = lcp then begin
17 |  |  if IsFirstPair(i) then begin
18 |  |  |  edges.append(pairs[lo].first) ; edges.append(pairs[lo].second)
19 |  |  |  lcp_array[i - 1] ? leaves.append(1) : leaves.append(0) ;
20 |  |  |  lcp_array[i + 1] ? leaves.append(1) : leaves.append(0) ;
21 |  |  |  visited[i] ← 1
22 |  |  |  count ← count + 2
23 |  |  else begin
24 |  |  |  edges.append(pairs[lo].second)
25 |  |  |  leaf_mark[i] ? leaves.append(1) : leaves.append(0) ;
26 |  |  |  visited[i] ← 1
27 |  |  |  count ← count + 1
   // push a sequence of count 0···01 structure bits
28 if lcp_array[i] = lcp then begin
29 |  PushStructureBits(count)
   // recurse on each proper subrange
30 for i in 1, ..., ranges.size() do
31 |  BulkConstruction(ranges[i].lo, ranges[i].hi, ranges[i].min_LCP)
```

To close the discussion about the bulk construction algorithm for the PT, we proof the worst-case running time bounds.

Theorem 3.7. *A Patricia Trie* PT *can be bulk constructed in $\mathcal{O}(|\mathcal{S}_{\mathsf{PT}}|)$ internal time, requiring no I/O operation if the corresponding array of mismatch pairs and LCP array are in main memory.*

After the transition from a subrange to a proper subrange, it brings about exactly one SA/LCP entry less to be processed, so that unfolding the trie from these arrays ends after probing $n = |\mathtt{SA}|$ array entries. This invariant stems from using the outcomes of the function $\mathtt{GetRanges}_{\mathsf{PT}}$, called at line 12, during the recursive calls in the loop of lines 30 and 31. From this invariant, we see that there will be no entry accessed more than once.

At this point, we demonstrated an algorithm which is theoretically suitable for the construction of tries out of the SA and LCP arrays and proved its theoretical efficiency regarding CPU processing time. As next, we combine this building block to the complete proposed String B-Tree's construction procedure and present some of its theoretical characteristics. From now on, we let the notations $\mathtt{BulkConstruction}_{\mathsf{PT}}$ and $\mathtt{UnfoldTrie}_{\mathsf{PT}}$ be used interchangeably.

3.5.2 String B-Tree

The deepest level of the String B-Tree is a layer of leaves and they are filled with pointers to the strings coming from the SA (or the sorted string array). We *copy* the rightmost string of each leaf to the next level of internal nodes and those strings are accumulated at the second level. Next step is to split them into blocks of size of an internal node, *extract* the rightmost string and push to the next level of internal nodes. So we do until we get a situation where the number of strings in a single level is less or equal the capacity of an internal node.

The number of I/O operations performed by the $\mathtt{BulkConstruction}_{\mathsf{SB}}$ operation is clearly dominated by the number of nodes n of the resulting String B-Tree, plus the I/Os used to access the text symbols. From the section 2 the B-Tree's height is bounded by $h \leq \log_B \frac{|\mathcal{S}|+1}{2}$. Observing the Algorithm 14 and assuming B as both the leaf node size and internal node size, without loss of generality we see that the deepest level pushes up $|\mathcal{S}|/B$ array entries to the upper String B-Tree level. Over there the rightmost string of each node is extracted to be pushed up recursively, as done in the previous step. Thus, the number of keys per level is always reduced by a factor of B. Summing up the number of internal nodes of each level, we get $n = |\mathcal{S}|/B + |\mathcal{S}|/B^2 + \cdots + |\mathcal{S}|/B^h = \sum_{i=1}^{h} \frac{|\mathcal{S}|}{B^i}$.

Theorem 3.8. *A String B-Tree* SB *can be bulk constructed with $\mathcal{O}(sort(|\mathcal{S}_{\mathsf{SB}}|) + scan(|\mathcal{S}_{\mathsf{SB}}|))$ I/O operations. The internal time is dominated by the time to construct the individual PTs, thus $\mathcal{O}(|\mathcal{S}_{\mathsf{SB}}|)$ internal work.*

Proof. To bulk construct a Patricia Trie, we firstly perform two random accesses to read out the SA and LCP arrays. Once this information is in main memory, we can bulk construct the Patricia Trie by unfolding it and letting the edge labels to be identified by their text positions generated by the SA and LCP arrays. This requires $scan(|\mathcal{S}_{\mathsf{SB}}|))$ I/Os to read the SA and LCP arrays.

To read out the text characters, rather than directly accesing the text symbols during trie unfolding, we sort all the edge label indices and use them to load the symbols after the unfolding phase with $\mathcal{O}(sort(|\mathcal{S}_{\mathsf{SB}}|)$ I/O steps.

After bulk constructing each PT, they must be also written into the external device, i.e. the internal nodes are subsequentially outputted as the corresponding PTs are generated. So, the

total number of I/Os done to write out the internal nodes is the total number of internal nodes, namely it is a geometric series, from which the worst-case running time of this step can be deduced:

$$\begin{aligned}
\sum_{i=1}^{h} \frac{|\mathcal{S}|}{B^i} &= |\mathcal{S}| \frac{1-(1/B)^h}{1/B} \\
&= |\mathcal{S}| \Big(B - B\Big(\frac{1}{B^h}\Big)\Big) \\
&= |\mathcal{S}| \Big(B - \frac{B}{\frac{|\mathcal{S}|+1}{2}}\Big) \\
&= B|\mathcal{S}| - \frac{2B|\mathcal{S}|}{|\mathcal{S}|+1} = \mathcal{O}(scan(|\mathcal{S}|))
\end{aligned} \tag{3.1}$$

Summing up all steps, we need $\mathcal{O}(sort(|\mathcal{S}_{\mathsf{SB}}|) + scan(|\mathcal{S}_{\mathsf{SB}}|))$ I/O operations in the worst-case. □

From Equation 3.1 one can still observe some hidden constants. Nevertheless, the developed algorithm is very suitable for overlapping I/O and CPU processing, as well as for parallelization, so that these constant terms can be soften in practice. Once the mismatch character pairs are in main memory, accompanied by the LCP values, multiple threads for constructing the PTs can be started and overlapped with an I/O thread used to output the internal nodes. Finally, the Algorithm 14 is shown in the following.

Algorithm 9: BulkConstruction$_{\texttt{SB}}$ operation

```
Input: Suffix Array sa, LCP Array lcp, Leaf Node Size leaf_size,
       Mismatching Character Pairs pairs
1  num_leaves ← 1 + (sa.size() − 1) / leaf_size
   // cache data for construction, i.e. pairs (T[i − 1], T[i]) for i ∈ {1, ..., sa.size() − 1}
2  pairs ← GetMismatchChars(sa, lcp)
   // determine the data going up to the first level of internal nodes
3  for i in 1, ..., num_leaves do
4      lo ← i × leaf_size
5      hi ← min(i × leaf_size + leaf_size − 2, max_index)
6      for i in lo, ..., hi do
7          leaf.keys[i − lo] ← sa[i]
8          leaf.values[i − lo] ← in_values[i]
       // index of in-range minimum LCP value corresponding to the internal node's label
9      min_idx ← lcp.GetMinInRange(lo, hi)
       // prepare the data for the first level of internal nodes of the String B-Tree
10     sa_up.append(sa[hi])
11     pairs_up.append(pairs[min_idx])
12     lcp_up.append(lcp[min_idx])
13     children_up.append(leaf)
   // recursively bulk construct the B-Tree level-by-level
14 BulkConstruction-Aux(1, sa_up, pairs_up, lcp_up, children_up)
```

Algorithm 10: `BulkConstruction`$_{\texttt{SB}}$ auxiliary function of the `BulkConstruction`$_{\texttt{SB}}$ operation

```
Input: Internal Node Size internal_size, Level's Height height,
       Upcoming Pairs of Mismatch Characters pairs, Upcoming Children Pointers children
       Upcoming Suffix Array Subpart sa, Upcoming LCP Array Subpart lcp
1  num_internals ← 1 + (sa.size() − 2) / internal_size
   // number of strings of the subset fits into a single internal node, so build the root
2  if sa.size() ≤ internal_size then begin
3      root.trie ← UnfoldTrie_PT(0, max_index, sa, lcp, pairs)
4      root.occupation ← sa.size()
5      for i in lo, ..., hi do
6          root.child[i] ← children[i]
7  else begin
       // construct the B-Tree for an intermediary level of internal nodes
8      for i in 1, ..., num_internals do
9          lo ← i × internal_size
10         hi ← min(i × internal_size + internal_size − 1, max_index − 1)
11         internal.trie ← UnfoldTrie_PT(0, max_index, sa,lcp, pairs)
12         internal.occupation ← internal.trie.size()
13         rightmost_idx ← hi − lo + 1
           // prepare the data for the first level of internal nodes
14         sa_up.append(sa[rightmost_idx])
15         pairs_up.append(pairs[rightmost_idx])
16         lcp_up.append(lcp[rightmost_idx])
17         children_up.append(internal)
   // recursively bulk construct the B-Tree level-by-level
18 BulkConstruction-Aux(height + 1, sa_up, lcp_up, pairs_up, children_up)
```

After describing the algorithms and proving running time guarantees, we give some notes on space complexity.

Lemma 3.9. *Let $PT_{\mathcal{S}}$ be a Patricia Trie serialized using the MA method and $PT_{\mathcal{S}_1}$ and $PT_{\mathcal{S}_2}$ the tries resulting from the* `Split`$_{\texttt{PT}}$ *operation. Regarding the size of the tries the following holds: $|PT_{\mathcal{S}_1}| + |PT_{\mathcal{S}_2}| \leq |PT_{\mathcal{S}}|$.*

Proof. The first case is when there is no space reduction and $|PT_{\mathcal{S}_1}| + |PT_{\mathcal{S}_2}| = |PT_{\mathcal{S}}|$. In this way, the `Split`$_{\texttt{PT}}$ operation doesn't induce any child-parent merge. Then we need to replicate the root and simply cut a subtree away from the original trie, yielding space usage of $|PT_{\mathcal{S}}|$, where $|PT_{\mathcal{S}_1}|$ is the space used for one subtree and $|PT_{\mathcal{S}_2}|$ is the space used for the second subtree, since root nodes don't occupy space in the MA method. The second case is when $|PT_{\mathcal{S}_1}| + |PT_{\mathcal{S}_2}| < |PT_{\mathcal{S}}|$ and it deals with single-childed nodes induced by splitting, whenever branches of a node lie on different tries after splitting, and one of the subtrees holds one node single-branched node. We only store the edge labels accompanied by two bits of structure information and a corresponding skip value. For each single-childed node we erase its edge label, two bits and one skip value. This yields an overall space consumption less than $|PT_{\mathcal{S}}|$. □

Theorem 3.10. *Any String B-Tree state requires less or equal space than a single external memory Patricia Trie, whenever using the space-optimal serialization method.*

Proof. The simplest case is when the String B-Tree is a leaf. In this case no Patricia Trie is needed and we have nothing to prove. From Lemma 3.9, any $\texttt{Split}_{\textsf{PT}}$ operation causes less space consumption. By property (2) from the Definition 2.4 of the B^+-Tree, each string is represented only once in the index. Thus, the space consumption is always reduced after splitting, otherwise we consume as much space as the external memory Patricia Trie, because the representation is the same. □

Theorem 3.11. *The bulk construction algorithm for String B-Trees derives a worst-case optimal layout for the String B-Tree structure in asymptotic optimal internal time and $\mathcal{O}(sort(|\mathcal{S}_{\textsf{SB}}|)+scan(|\mathcal{S}_{\textsf{SB}}|))$ I/Os, occupying less space than a single external memory Patricia Trie, whenever using the multiarray serialization method.*

The layout/fragmentation aspects of the Theorem 3.11 are results similar to the ones from [14], as the space consumption guarantees come from the Theorem 3.10 and the discussion in the subsection 3.4.1. Since external memory PTs built over all suffixes of a string are much more compact than any ST, we can state that our String B-Tree can be the smallest full-text index available.

In comparison to the SA accompanied by the LCP array, one String B-Tree needs extra space for holding the edge labels and the string pointers within the PTs. Nevertheless it can be configured not to store some LCP entries that the LCP array does, so that the String B-Tree can also be a strong competitive against the SA and LCP arrays in practice. Further discussions are shifted to the section 5, which discusses the practical experiments.

Complexity Summary

Algorithm	**Internal Work** (iterations)	**External Work** (I/Os)
Patricia Trie PT	$\mathcal{O}(\|\mathcal{S}_{\textsf{PT}}\|)$	$\mathcal{O}(1)^*$
String B-Tree SB	$\mathcal{O}(\|\mathcal{S}_{\textsf{SB}}\|)$	$\mathcal{O}(sort(\|\mathcal{S}_{\textsf{SB}}\|) + scan(\|\mathcal{S}_{\textsf{SB}}\|))$

Table 6: Worst-case summary table for the bulk construction algorithms

* If the mismatching pairs and LCP values are cached in main memory or saved in a request schedule containing the addresses of the edge labels' characters and skip values for posterior fetching.

3.6 Limitations

String B-Trees are restrictive because of long execution times for construction, since most approaches are based on iteratively constructing the index. We overcome this limitation by reusing the SA and LCP arrays to lift up a supra-index with much less I/O steps. Still the preprocessing time for generating the SA and LCP arrays is required, but it is also a preprocessing step in many efficient algorihtms for constructing the STs [27] and it runs very fast for large input data sets.

The implementation of the internal node's dynamic operations is also not trivial, since heuristics are envolved. Usual implementations of the String B-Tree rely on packing and unpacking the

internal nodes whenever they are loaded from the external memory and this can be prohibitive in terms of execution time and restrictive regarding space consumption, because of the required pointers.

We noticed that the range of applications of STs [7] must not necessarily be supported by the String B-Trees. It comes out that its B-Tree structure can lead to more complicated algorithms or let the problems be not possible to solve with the String B-Trees. There is no concrete study about the feasibility of all those ST applications and it is an open question, whether further algorithms can be developed for the String B-Tree, other than the ones used for solving the *substring problem*. This makes difficult to use the String B-Tree data structure for some advanced applications.

4 Implementation Details

We implemented all the algorithms and data structures using the C++ programming language, because of its flexibility and multiparadigm approach. The source-code is freely available for reproduction of results. From this point on, we use the terms succint representation and multiarray representation to denote the same data structure serialization approach. `byte` refers to the *byte-base* serialization method and `succ` refers to the multiarray serialization method.

To better evaluate conditional expressions, the external memory address zero is reserved, so that addressing starts from one. To allow for better reuse of code, while fulfilling performance gurantees, the key storage is propagated from the String B-Tree class to the PT class via a reference passed as constructor parameter. The PT class is nested as a template parameter of the String B-Tree class, so that all internal nodes have the same type.

4.1 String B-Tree

We adopted a template-based implementation for sake of reuse and code simplification. To allow for different models of computation, different storage models are represented by different classes. Thus, data stored in the main memory are accessed from the program in exactly the same way as it is accessed whenever stoered in the external memory device. To switch between the different models of computation, we use preprocessor directives. It follows that the constructor parameters for the different storage models are different, so that a template parameter would not be the correct option.

To allow for easy adaptation to multiple models, we address nodes with an `size_t` memory word. It can be used either as an external memory offset or a main memory pointer and reinterpreted according to the compiling directives. This allows not only for a better reuse in the context of RAM and DAM algorithms, but also the network algorithms can be easily adapted. Indeed encapsulating those addresses into a class would be an interesting idea, so that the adaptation to the network model can be directly done.

Allocators allowed for a better memory management, so that the data structure demands can be properly fulfilled whenever a node must be used, independently of its type (an internal node or a leaf node). This combinated with the storage for keys and values allow for a complete customization of the memory management.

4.2 Patricia Trie

A boolean compiler directive is given to signal whether the PT has its own key storage object managed as member variable or it uses a reference from a client object. The swithcing between DAM and RAM implementations are done in the same fashion as done by the String B-Tree implementation.

In implementing the algorithms of the multiarray serialization method, we added a dummy array entry to the first position of the E_{MA}, S_{MA}, and T_{MA} arrays to allow for a simpler implementation of the $\texttt{Insert}_{\texttt{PT}}$ operation. Those dummy symbols are used to represent a fictitious incoming edge to the root node, so that the root node can be processed in the same way as the other PT nodes are processed. This dummy value is actually conceptual and can be removed, accounting for short adaptations of the algorithms. The gains of removing the dummy symbol are negligible, so that obtaining a simpler implementation worths more than making the algorithms a little bit more complicated to save few bits of data representation.

5 Experimental Results

We implemented the String B-Tree and the Patricia Trie data structures. The PT can be used as a standalone program or be plugged to the String B-Tree as representation of the internal node. Doing so allows for better evaluation and reuse, so that the PT code can be reused for other purposes. Additionally, tests were implementing using a Unit framework to check correctness for different parameters and variants. In order to fix some memory management bugs, we reproduced the tests in separated applications and used the UNIX tool *valgrind* to find memory leaks and malicious computations. All the code, tests and some test data is avaiable at `http://www.github.com/fblima`.

5.1 Experimental Setup

To evaluate the PT implementations, we compare the two serialization methods discussed in this thesis, namely the byte-based (`byte`) and the multiarray (`succ`) methods, using different input sizes. More exactly, the operations $\texttt{Search}_{PT}$, $\texttt{Insert}_{PT}$ and $\texttt{BulkConstruction}_{PT}$ are compared in terms of space usage and CPU running time. For sake of simplicity, the evaluation of the String B-Tree implementation is done using solely the multiarray method to represent the internal nodes, since using the `byte` variant yields results with the number of IO's.

In all our experiments we used an Intel(R) Xeon(R) CPU X5355 @ 2.66GHz with approximately 24 GiB of main memory. The operating system running was Ubuntu over the kernel 3.13.0-39-generic.

5.2 Input Data Sets

To evaluate the performance of the data structure, input strings are generated *uniformly* at random. For the evaluation of the $\texttt{Insert}_{PT}$ operation of both data structures, the string content and the string length are taken at random.

Because of the inefficiency of the $\texttt{Insert}_{SB}$ operation we restricted the string lengths to be at most 1,000 and varied the number of strings over bigger ranges. Conversely, the experiments done with the PT don't assume any restriction about the string lengths, since these algorithms are designed to run in the main memory.

5.3 Discussion

5.3.1 Patricia Trie

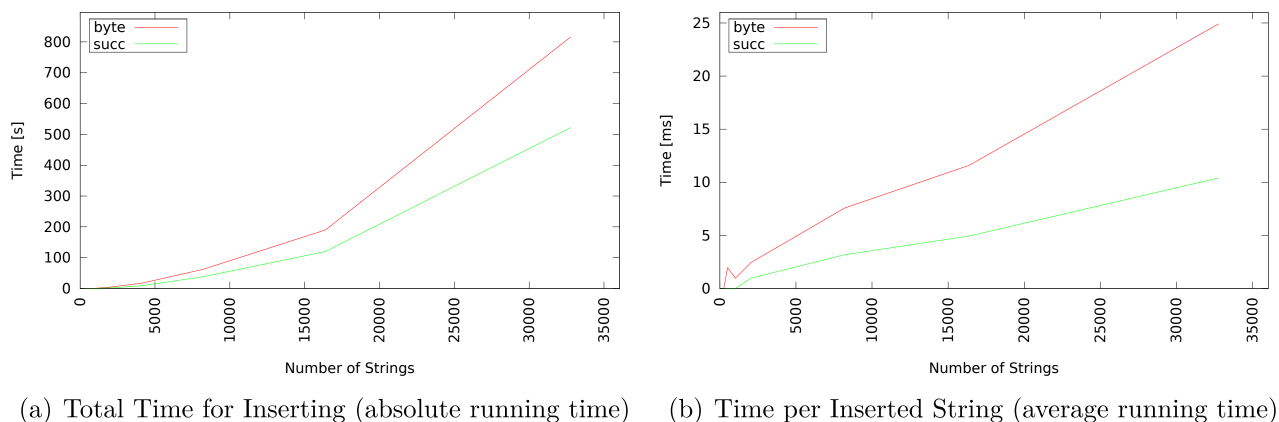
The different serialization approaches allow for the tradeoff of performance for space consumption of the String B-Tree's internal node. The `succ` variant is space-optimal and further space usage improvements could be accomplished using compression, however it would account for slower $\texttt{Insert}_{PT}$ and $\texttt{Search}_{PT}$ running times, since the data must be compressed/decompressed after/prior to usage. In this thesis we focus on the theoretical worst-case evaluation of the PT serialization methods and postpone the entropy compression optimizations for the future works.

Execution Time

The Figure 32 presents the performance of the $\texttt{Insert}_{PT}$ operation. As predicted, the operation running time is strongly related to its space consumption, thus linearity can be clearly recognized. Observing the absolute measures, we can realize that the algorithms running on both data representations start to get slower for input sizes between 15,000 and 20,000. During our experiments, we could settle that this is the point where the memory used for representing the data structure and the memory occupied for the state of the algorithms plus operating system data structures, starts to exhaust the available free main memory, so that pages must be swapped between main memory and the external memory device. At this stage, paging is controlled by the operating system, so that only efficient external memory algorithms could cause performance improvements and avoid this significant performance loss.

From the average running time numbers of Figure 32(b), we can conclude that the multiarray variant is not only faster, but offers a much more predictable worst-case running time. We observe an absence of peaks yielding a smooth curve, what usually indicates regularity on the performance.

We conclude that the multiarray method is faster and more stable for insertions than the byte-base serialization.



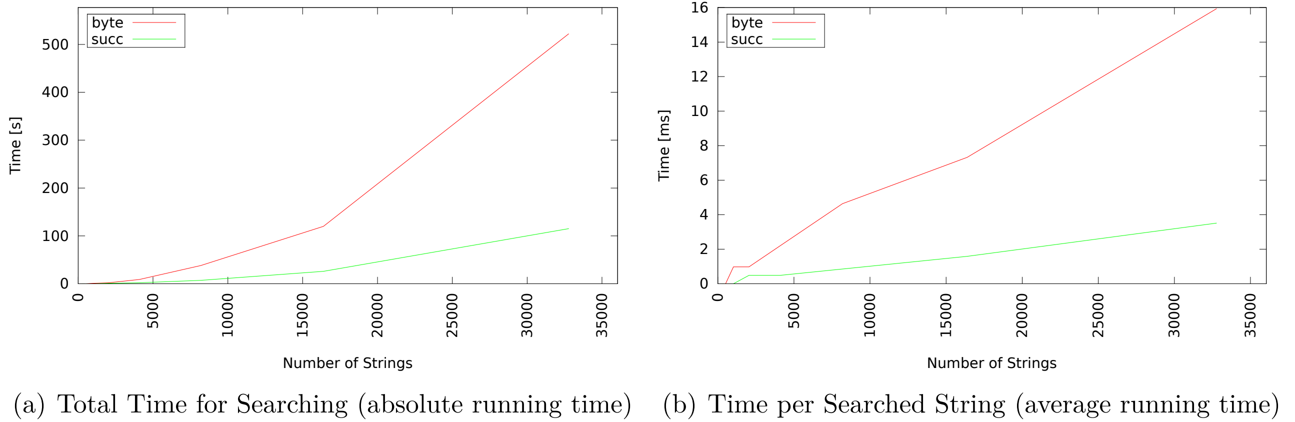
(a) Total Time for Inserting (absolute running time) (b) Time per Inserted String (average running time)

Figure 32: Analysis of the $\texttt{Insert}_{PT}$ operation

A straightforward comparison between the figures 32 and 33 shows that both operations present very similar asymptotic running times. The reason is that the first step of the $\texttt{Insert}_{PT}$ operation is actually the $\texttt{Search}_{PT}$ operation, which is followed by a trie transformation that shifts big memory chunks, what accounts for the longer running times of the $\texttt{Insert}_{PT}$ operation.

Nevertheless, it worths to notice that searching using the data structure serialized by the multiarray method asymptotically slows down much less than using the byte-based approach. The reason is that the multiarray serialization requires much less space and it was evaluated using the jump pointers to perform the `GetChild` atomic operation.

We can also observe that searching using the byte-based method has an abrupt slow down for very small input sizes, as observed from the charts regarding the $\texttt{Insert}_{PT}$ operation. From that we can realize that it is an irregularity specific to the data representation. The economic bits of the multiarray representation allow for less side effects at the beginning, since the data structure doesn't grow very quickly at that point.



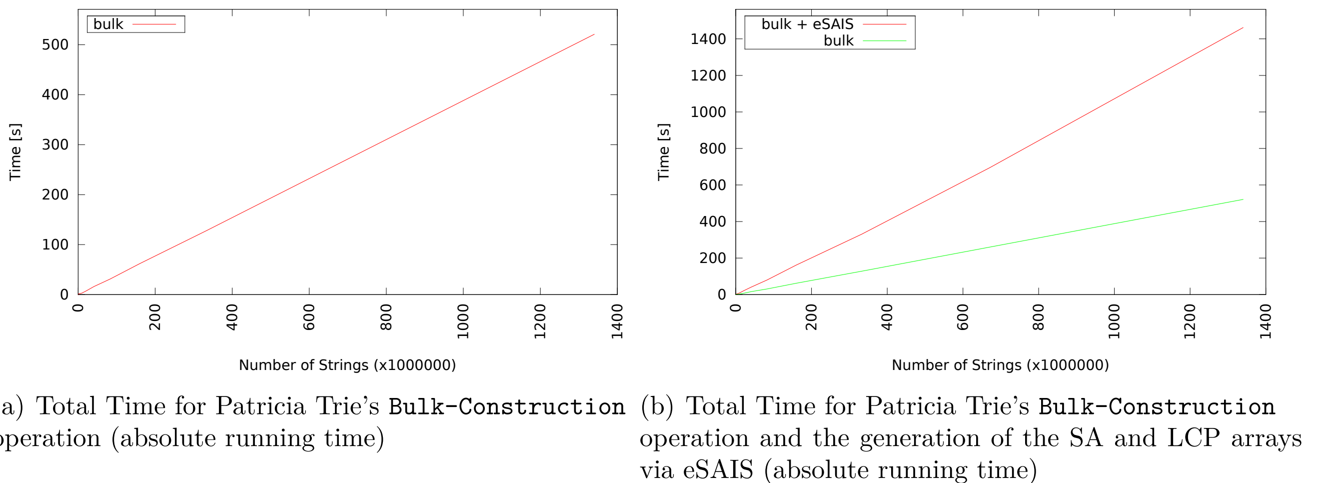
(a) Total Time for Searching (absolute running time) (b) Time per Searched String (average running time)

Figure 33: Analysis of the $\texttt{Search}_{\texttt{PT}}$ operation

The next charts confirm our prediction from analysis, that the PT's bulk construction algorithm runs in linear internal time, even though we use amortization in our theoretical argumentation. The Figure 34(a) shows the absolute running time solely for unfolding the trie from the SA and LCP arrays, as the Figure 34(b) includes the execution time for generating the SA and LCP arrays using the eSAIS algorithm [47]. Comparing both steps one can clearly realize that both are definitively linear and their performance can be precisely predicted.

In this experiment we unfold the trie directly onto the multiarray representation and don't measure the process for unfolding a byte-based representation. It follows that the multiarray data structure yields much simpler algorithms and this representation directly reflects the outcomes resulting from the traversal of the SA and LCP arrays using the algorithm presented in this thesis.

Comparing these numbers with the running time measures offered by the iterative construction method, namely by using the $\texttt{Insert}_{\texttt{PT}}$ operation, we see that the combination of eSAIS and our bulk construction procedure perfectly outperforms the naive iterative procedure regarding execution time.



(a) Total Time for Patricia Trie's `Bulk-Construction` operation (absolute running time) (b) Total Time for Patricia Trie's `Bulk-Construction` operation and the generation of the SA and LCP arrays via eSAIS (absolute running time)

Figure 34: Analysis of the $\texttt{BulkConstruction}_{\texttt{PT}}$ operation

To clearly point out that the bulk construction algorithm definitively scales well and the mul-

tiarray representation offers the best deal regarding both $\texttt{Insert}_{PT}$ and $\texttt{Search}_{PT}$ operations, the Figure 5.3.1 summarizes the algorithmic evolution introduced by using the algorithm engineering ingredients introduced in the past chapters.

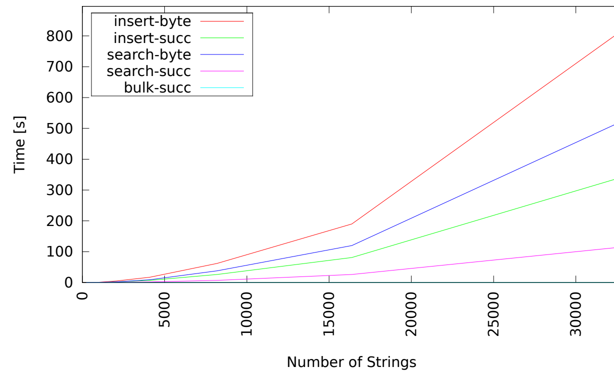
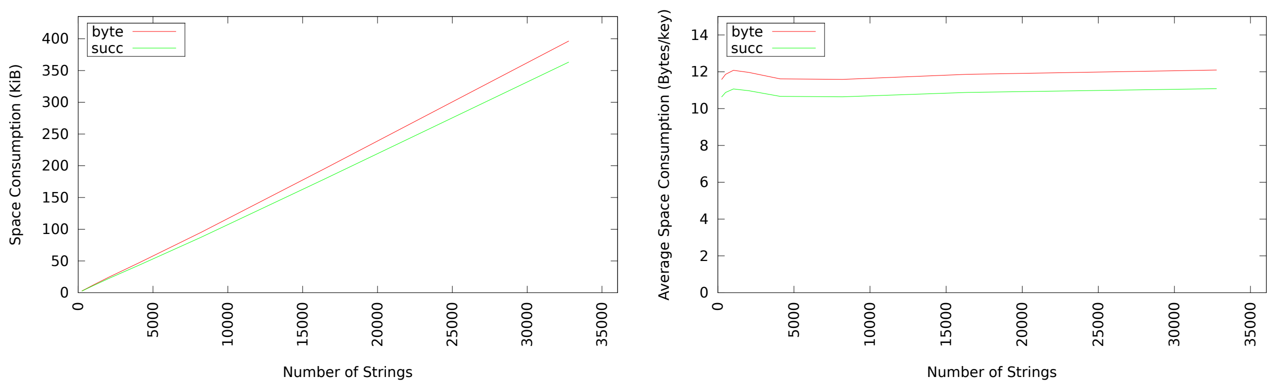


Figure 35: Analysis of the Patricia Trie operations

Space Consumption

During analysis we deduced linearity on the number of stored keys for space consumption, as summarized in the Table 3, and indicated the space consumption growths per newly inserted keys in the Table 4. The Figure 36(a) with the absolute space consumption measures can be used to confirm this linearity, as the Figure 36(b) experimentally proves that the multiarray variant consumes less space than the byte-based method. In the later chart, one can also observe a little irregularity for small instances caused for the trie's bootstrap growth. Recalling the Figures 32 and 33, we see that although both serialization methods have the same bootstrap phenomena regarding the average space usage, the multiarray variant shows much less effects regarding running time performance due its engineered memory usage and data representation.



(a) Overall Space Consumption (absolute space usage) (b) Bytes per Input String (average space usage)

Figure 36: Analysis of the Patricia Trie's space consumption

Summarizing, our measures show that the multiarray representation is very space efficient and can be recommended for main memory applications.

5.3.2 String B-Tree

There are at least three String B-Tree parameters that could be varied to make it better to understand the data structure's dynamics. One could systematically vary the serialization approach and the internal and leaf nodes sizes.

In our experiments, the `succ` serialization method is the unique being used. This parameter doesn't influence the B-tree layout in our implementation, since we use the number of keys as the filling measure for the internal nodes. Using the number of keys, and not the size in bytes, is reasonable, because PTs have a limited number of internal nodes, which increases linearly with the number of stored keys. Furthermore, from the first experiments, the String B-Tree showed to be definitively I/O bound, so that varying the internal node representation doesn't have relevant importance.

The internal and leaf sizes are important parameters, which can be varied according to the target application. Chosing larger leaf node sizes account for a more compressed index, since less keys must be represented along the internal nodes. Taking smaller leaf nodes implies on bigger indexes, but less binary search steps must be done during searching. The best space-time deal is a matter of doing calculations, experimenting the String B-Tree and chosing it according to the target application. For simplicity, we restrict our experiments to a setup where both leaf and internal node sizes are defined to be 10% of the input size. From the practical experiments, 10% showed to offer good results, where we are able to analyze the most important aspects of the String B-Tree regarding the running time.

To present the results of the experiments of the String B-Tree, we ommited the charts containing the average numbers because they didn't reveal interesting aspects regarding this experiment setup. Adding more charts would make the text redundant, so that the reader is encouraged to used our implementations to generate further numbers and study further practical issues, like to experiment different node sizes and to configure the data structure to be faster than other approaches.

Execution Time

We measured the number of I/Os and absolute running times of the bulk construction algorithm as well as the performance of the iterative constructin algorithm using the DAM storage model. In these experiments, the number of I/O operations regards the number of saved and loaded String B-Tree nodes. The I/Os involving string comparisons are not counted, because they are a point of optimization and don't reveal interesting aspects of the data structure.

From the Figure 38, we observe that the number of performed I/Os by all operations directly depends on the input size. The key result is that our bulk construction method is rather faster than all other operations, since it roughly loads the arrays and quickly perform operations on that.

The $\texttt{Search}_{\texttt{SB}}$ operation needs much less I/O operations than the $\texttt{Insert}_{\texttt{SB}}$ operation, because it simply traverses the String B-Tree and don't induce any node splitting. Further improvements could be done whenever better designing the internal and leaf node sizes.

The absolute running time is another important measure, since it defines the quality of the resulting data structure from a client perspective. We observe that all the operations have the same asymptotic behavior, as expected. Again, our bulk construction method is confirmed to be faster than all other operations. Asymptotically, all other operations show the same growth as the I/O analysis of Figure 38 shows.

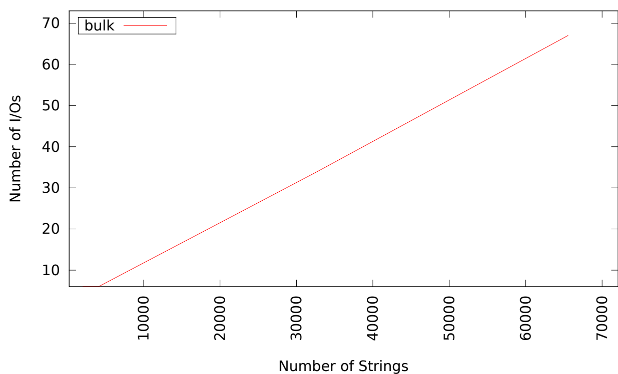
However, interestingly the $\texttt{Search}_{\texttt{SB}}$ operation seems to be not so slow as the numbers of Figure 37(d) show. The reason is that the string comparisons were not counted during the experiments

depicted in the Figure 37(d) and we are also performing linear search in the leaves, what makes searches much slower. That contributes a lot for the running time and an important performance improvement would be to carefully design this procedure, firstly by using a binary search and then by avoiding unnecessary character comparisons or using hashing.

Space Consumption

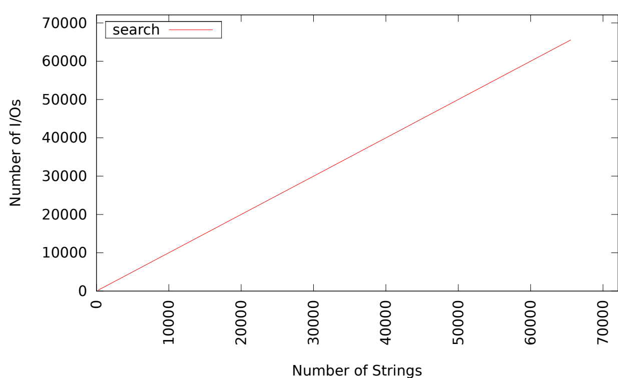
The space consumption of our String B-Tree data structure corresponds to our expectations, that depending on how the leaf and internal node sizes are chosen, much less space is needed for the index in comparison to a single PT holding the same string set. The String B-Trees hold no tree information at the level of the leaves and that contributes a lot for the space savings.

As mentioned before, we use 10% of the input size as the node size and this implies that for any given input size, the space consumption is nearly the same. Thus, with this ratio the index grows in a nearly constant way, as depicted in the Figure 39.

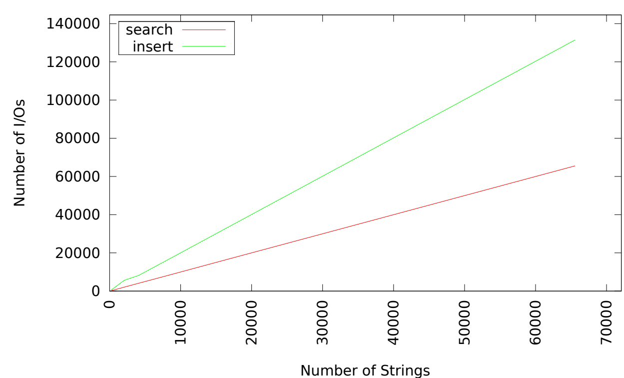


(a) Total Number of I/O operations of the String B-Tree's $\texttt{BulkConstruction}_{SB}$ operation

(b) Total Number of I/O operations of the $\texttt{Insert}_{SB}$ operation

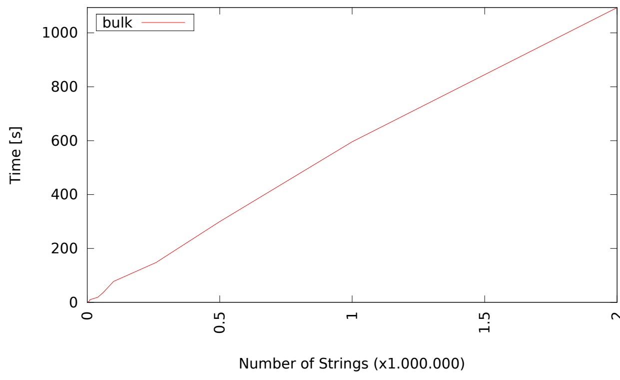


(c) Total Number of I/O operations of the $\texttt{Search}_{SB}$ operation

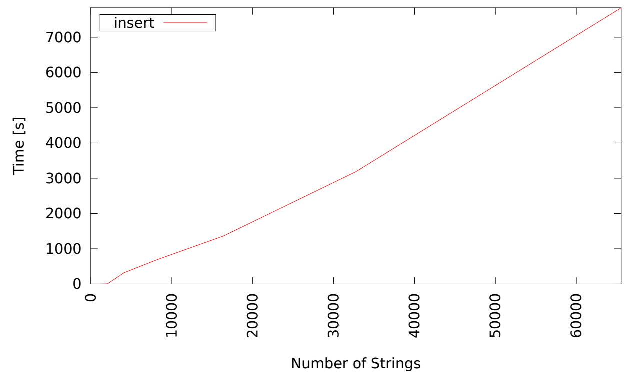


(d) Total Number of I/O operations of the $\texttt{Insert}_{SB}$ and $\texttt{Search}_{SB}$ operations

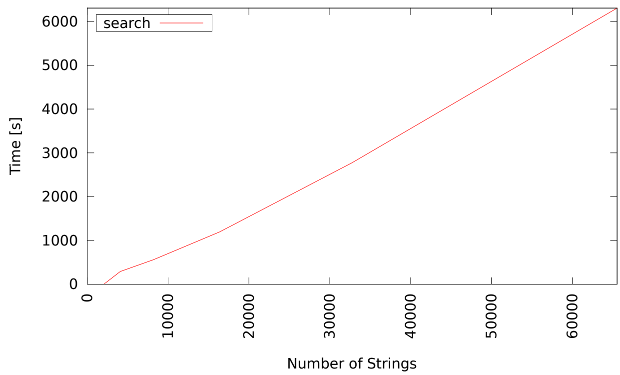
Figure 37: Analysis of the String B-Tree regarding the absolute number of I/Os



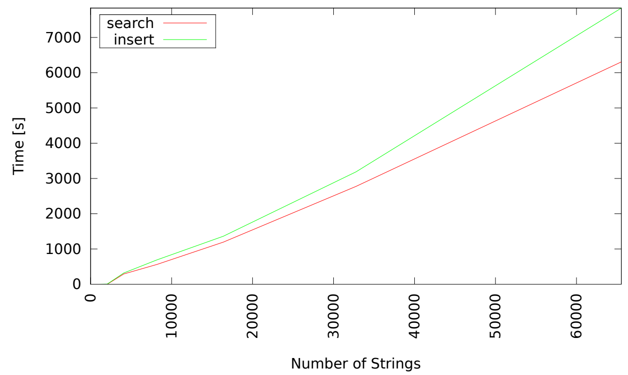
(a) Total Time for the BulkConstruction$_{SB}$ operation (absolute running time)



(b) Total Time for the Insert$_{SB}$ operation (absolute running time)



(c) Total Time for the Search$_{SB}$ operation (absolute running time)



(d) Total Time for the Insert$_{SB}$ and Search$_{SB}$ operations (absolute running times)

Figure 38: Analysis of the String B-Tree regarding the absolute running time

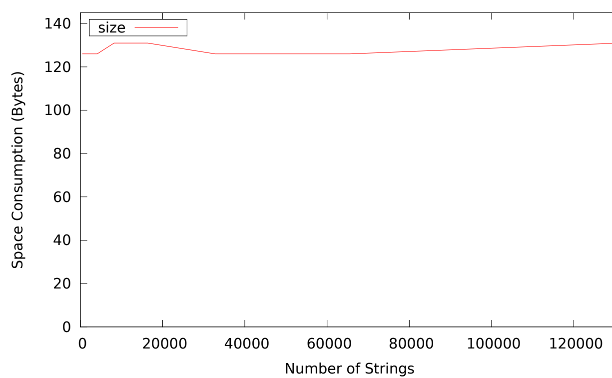


Figure 39: Analysis of the String B Tree's space consumption

6 Conclusion

In this thesis we worked with a practical approach to the String B-Tree due Ferragina [34]. This means reviewing the B-Tree's definition, designing space-optimal representations of the String B-Tree's internal node, that have practical and theoretical performance gurantees, and elaborating a construction process with efficient worst-case running time regarding CPU processing time and I/O operations.

At the practical part, we focused on generic programming in order to allow for posterior code reusage and easy integration with other systems. We tested our implementations using assertions, unit tests for checking the correctness of the operations and evaluated space and time performance of the algorithms. At the bottomline, the String B-Tree turned out to be feasible for usage in practice, since our construction running time is not absurdly greater if compared to the ones provided by open implementations.

6.1 Future Work

It would be interesting to evaluate the String B-Trees as an underlying model for network applications and check whether its structure can be used to fulfill the needs of distributed systems. Despite of similarity to the $\texttt{Insert}_{\texttt{SB}}$ operation, the $\texttt{Delete}_{\texttt{SB}}$ operation was not implemented and it would be interesting to check it whether the underlying atomic operations yield further interesting problems.

In our implementations we focused on getting to know the theoretical limits of our data structures, that is we implemented a slow version of the byte-based serialization method, which uses no extra pointers for navigation, and a fast version of the multiarray serialization method, which uses extra jump pointers for navigation. The former has the practical meaning that it is not as slower as the naive data structure implementation, but tends to be less efficient than the mutliarray algorithms with jump pointers. On the other hand the fast implementation of the mutliarray (succint) serialization method represents the other extrem, where the data structure is stored using as less bits as possible without using any compression technique. An interesting further task would be to evaluate other variants, like a fast byte-based implementation with jump pointers, or similarly a slow multiarray implementation without jump pointers. Doing so would help database designers to establish tradeoffs and to experience for what input sizes it worths to switch between different data representations.

As mentioned in the beginning, the distributed approach would be definitively interesting to check out. Different remote procedure call (RPC) frameworks could be tested and evaluated with different algorithm variants. An interesting practical issue would be to evaluate our algorithms using different hardware architectures, in order to identify tuning parameters. Furthermore, entropy compression techniques could be investigated. As saw in the exeperiment results, the String B-Tree data structure is I/O bounded, what could make room for longer internal work while overlapping CPU processing and IO operations. Reproducing those experiments using Solid State Drives (SSDs) is also another important aspect, so that the gap between technology and algorithmics can be more exactly defined. It could be that SSDs let the data structure be CPU bounded, so that parallelization turns out to be mandatory to accomplish best-effort performance outcomes.

Another interesting aspect would be to evaluate running time using different values for the data structure parameters, namely checking how the running time varies according to different internal and leaf node sizes. Also important would be to compare the String B-Tree to alternative full-text data structures, like the STs, regarding its applicability and performance.

References

[1] A. Blumer, J. A. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas: *The smallest automaton recognizing the subwords of a text.* 1985.

[2] Andreas Crauser and Paolo Ferragina: *A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory.* 2002.

[3] Anil Maheshwari and Norbert Zeh: *A Survey of Techniques for Designing IO-Efficient Algorithms.* 2003.

[4] Benjarath Phoophakdee and Mohammed J. Zaki: *Genome-scale disk-based suffix tree indexing.* 2007.

[5] D. R. Clark and J. I. Munro: *Efficient suffix trees on secondary storage.* 1996.

[6] Daan Leijen; Wolfram Schulte; Sebastian Burckhardt: *The design of a Task Parallel Library.* 2009.

[7] Dan Gusfield: *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1 edition, 1997.

[8] David Weese: *Entwurf und Implementierung eines generischen Substring-Index.* 2006.

[9] Diego Arroyuelo, Carolna Bonacic, Veronica Gil-Costa, Mauricio Marin and Gonzalo Navarro: *Distributed Text Search using Suffix Arrays.* 2014.

[10] Donald E. Knuth: *Sorting and Searching, volume 3 of The Art of Computer Programming.* Addison-Wesley, 2 edition, 1971.

[11] Donald R. Morrison: *PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric.* 1968.

[12] Douglas Comer: *The Ubiquitous B-Tree.* 1979.

[13] E. M. McCreight: *A space-economic suffix tree construction algorithm.* 1976.

[14] Erik D. Demaine, John Iacono, Stefan Langerman: *Worst-Case Optimal Tree Layout in External Memory.* 2013.

[15] Essam Mansour, Amin Allam, Spiros Skiadopoulos and Panos Kalnis: *ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings.* 2011.

[16] Felipe a. Louza, Guilherme P. Telles and Cristina Dutra de Aguiar Ciferri: *External Memory Generalized Suffix and LCP Arrays Construction.* 2013.

[17] Ge Nong, Sen Zhang and Wai Hong Chan: *Linear Suffix Array Construction by Almost Pure Induced-Sorting.* 2009.

[18] Gerth Stølting Brodal and Rolf Fagerberg: *Cache-Oblivious String Dictionaries.* 2006.

[19] Gonzalo Navarro, João Paulo Kitajima, Berthier Araújo Ribeiro-Neto and Nivio Ziviani: *Distributed generation of suffix arrays.* 2005.

[20] Johannes Fischer and Volker Heun: *A New Succint Representation of RMQ-Information and Improvements in the Enhanced Suffix Array.* 2007.

[21] Juha Kärkkäinen and S. Srinivasa Rao: *Full-Text Indexes in External Memory.* 2003.

[22] Juha Kärkkäinen, Peter Sanders and Stefan Burkhardt: *Linear Work Stuffix Array Construction.* 2006.

[23] Kunihiko Sadakane: *Compressed Suffix Trees With Full Functionality.* 2007.

[24] Manuel Scholz: *Experimentelle Untersuchung von String B-Trees bezüglich ihrer Anwendbarkeit in Genom-Datenbanken und im Information Retrieval.* 2002.

[25] Marina Barsky, Ulrike Stege, Alex Thomo and Chris Upton: *A New Method for Indexing Genomes Using On-Disk Suffix Trees.* 2010.

[26] Martin Farach-Colton: *Optimal Suffix Tree Construction with Large Alphabets.* 1997.

[27] Martin Farach-Colton, Paolo Ferragina and S. Muthukrishnan: *On the Sorting-Complexity of Suffix Tree Construction.* 2000.

[28] Masahiro Kasahara and Shinichi Morishita: *Large-Scale Genome Sequence Processing.* Imperial College Press, 1 edition, 2006.

[29] Matteo Comin and Montse Farreras: *Efficient parallel construction of suffix trees for genomes larger than main memory.* 2013.

[30] P. Weiner: *Linear pattern matching algorithm.* 1973.

[31] Paolo Ferragina and Roberto Grossi: *Fast incremental text editing.* 1995.

[32] Paolo Ferragina and Roberto Grossi: *A fully-dynamic data structure for external substring search.* 1995.

[33] Paolo Ferragina and Roberto Grossi: *An Experimental Study of SB-Trees.* 1996.

[34] Paolo Ferragina and Roberto Grossi: *The String B-Tree: A New Data Structure for String Search in External Memory and its Applications.* 1998.

[35] Paolo Ferragina and Roberto Grossi: *Simple Implementation of String B-Trees.* 2004.

[36] Paolo Ferragina and Rossano Venturini: *Compressed Cache-Oblivious String B-Trees.* 2006.

[37] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini and S. Muthukrishnan: *Structuring labeled trees for optimal succinctness, and beyond.* 2005.

[38] Peter Sanders: *Memory Hierarchies - Models and Lower Bounds.* 2003.

[39] Raphael Clifford: *Distributed Suffix Trees.* 2004.

[40] Raphael Clifford and Marek Sergot: *Distributed and Paged Suffix Trees for Large Genetic Databases.* 2003.

[41] Ricardo Baeza-Yates and Berthier Ribeiro-Neto: *Modern Information Retrieval.* ACM Press, 1 edition, 1999.

[42] Ricardo Baeza-Yates, Eduardo Fernandes Barbosa and Nivio Ziviani: *Hierarchies of Indices for Text Searching.* 1996.

[43] Richard Cole, Tsvi Kopelowitz and Moshe Lewenstein: *Suffix Trays and Suffix Trists: Structures for Faster Text Indexing.* 2013.

[44] Rossano Venturini: *On searching and extracting strings from compressed textual data.* 2004.

[45] Stefan Kurtz: *Reducing the Space Requirement of Suffix Trees.* 1998.

[46] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein: *Introduction to Algorithms.* The MIT Press, 2 edition, 2001.

[47] Timo Bingmann, Johannes Fischer and Vitaly Osipov: *Inducing Suffix and LCP Arrays in External Memory.* 2009.

[48] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa and Kunsoo Park: *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applciations.* 2001.

[49] U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* 1993.

[50] Xingyan Fan, Yang Yang, Liang Zhang: *Implementation and Evaluation of String B-Tree.* 2001.