An experimental study of SB-trees *

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
ferragin@di.unipi.it

Roberto Grossi
Dipartimento Sistemi e Informatica
Università di Firenze
via Lombroso 6/17, 50134 Firenze, Italy
grossi@dsi2.ing.unifi.it

June 1996

Abstract

In a previous work of ours [13], we proposed a text indexing data structure for external memory, which we called SB-tree, that combines the best B-tree and suffix array qualities to overcome the limitations of inverted files, suffix arrays, suffix trees, and prefix B-trees. In this paper, we study the performance of SB-trees in a practical setting by running a large number of searching and updating experiments. We obtain fast practical performance by means of a new space-efficient and alphabet-independent organization of SB-tree nodes and a new batch insertion procedure that avoids thrashing.

1 Introduction

Textual data in electronic form are more available than before and range from published documents (e.g., electronic dictionaries, libraries and archives, etc.) to private databases (e.g., marketing information, legal records, medical histories, etc.). Online providers of legal and newswire texts (such as Westlaw and Lexis-Nexis) already have hundreds of text gigabytes and will have terabytes in the near future. Many texts are sent over Internet every day in the form of email, bulletin boards, web pages, etc.. Information overload is therefore a general problem because all these data need to be stored, updated and fast-accessed in external storage devices (e.g., disks, CD-ROMs) and this inevitably introduces the need to develop efficient text indexing data structures. Text indexing also has several applications to large text collections that can change over time and need some nontrivial search operations (e.g., data compression [34, 35, 9, 24], computer virus detection [18], text editing, telephone directory storage [7] and software maintenance [4]). Below, we discuss few of them:

Textual databases. In information retrieval environments, it is often necessary to deal with large heterogeneous text collections, such as newspaper files, dictionaries, encyclopedias, telephone

*This work is partially supported by MURST of Italy and was carried out while the authors were visiting the AT&T Bell Laboratories, Murray Hill, NJ. Part of the results had been previously presented at the *ACM-SIAM Symposium on Discrete Algorithms*, 1996.

directories, textbook materials, etc. The keywords are assigned to the information items manually or automatically and the queries are formulated by means of terms interconnected by boolean operators. In this context, indexing data structures and searching engines are fundamental tools for getting useful information from this (text) data. More in general, data structures for indexing textual databases also relate to the problem of indexing commercial databases. In this context, records are sequences of bytes that change over time and, consequently, a record can be interpreted as an arbitrary long text over a 256-character alphabet. Text indexing tools can therefore be essential for achieving efficiency and a large range of requests in many practical applications that handle large text collections.

Molecular biology. Industrial exploration has become a fundamental tool in biosciences. In molecular biology, for example, DNA sequencing has proven to be reliable in analyzing genetic information. The current volume of nucleic acid sequences in public databases amounts to approximately 200 million bases. Due to the rapid increase in the flow of information, some significant public genome databases have been established [15]. We can say that, in general, 'biocomputing' has become essential and thus indexing data structures and searching engines will play more and more an important role in sequence analysis research [3].

Digital libraries. A library's task is to acquire works, store them and make them available to the reader. In a recent issue of Communication ACM [14], many of the projects that are underway for the creation or enhancement of digital libraries are described. An example of this can be found in project Dienst [14, pag. 47]; its method consists in distributing the indexes and in processing the searches in parallel across each index site. Moreover, it allows users to search for documents by their number, title, author, etc., by entering the *text part* to be searched or by selecting a *paragraph* directly from a document with the mouse as the basis of the search. As a result, the creation of indexing data structures and searching engines for bibliographic material adds value to the material itself and greatly increases the accuracy and completeness of subsequent retrievals.

Consequently, there is a very large request for indexing data structures and searching engines in those fields. These tools differ from the ones designed for main memory because of current technology. Specifically, over the past fifteen years, disk drive access time has improved very little while memory densities have increased at an average of 50 percent a year, and memory access times have decreased from 30 to 80 percent a year [27]. Nevertheless, we need external storage devices because we cannot build any main memory with an unbounded capacity and single-cycle access time. For this reason, there is ongoing research to improve the I/O subsystem by introducing several hardware mechanisms such as disk arrays, disk caches, etc. [27]. On the other hand, since main memory is a high-speed *electronic* device and external memory is a relatively low-speed *mechanical* device, all the data must be suitably arranged on disks by using efficient (external-memory) data structures that minimize the number of I/Os [31]. A great deal of current research is directed at this as far as both information retrieval and web community are concerned (see for example: `http://inktomi.berkeley.edu` and `http://glimpse.cs.arizona.edu` for some impressive search facilities).

In this paper, we focus our attention on algorithms and data structures for searching on *arbitrarily-long text strings* stored in external memory with provably good performance. This is a hot topic nowadays, especially because there are very few *efficient* methods for extracting useful information from large (text) data. We can formalize our problem as follows:

Problem 1 Let $\Delta = \{\delta_1, \ldots, \delta_k\}$ be a set of text strings stored in external memory, whose total length is $N = \sum_{i=1}^{k} |\delta_i|$ characters. We can change Δ *dynamically* by inserting or deleting some

individual strings and we can search on-line for all the *occ* occurrences of an arbitrary pattern $P[1, p]$ in Δ's strings. [1]

Unfortunately, the elegant external-memory data structures dealt with in current literature either lack good searching bounds in the worst case or are unable to support efficient dynamic operations. For example, in the area of traditional external-memory data structures, inverted files [28], B-trees [5] and their variations (such as Prefix B-trees [6, 10]), well-known and ubiquitous tools for manipulating large data, are not as good as they could be for solving Problem 1. Inverted files require sublinear space but are difficult to update. B-trees help to overcome this drawback but they only treat *bounded-length* keys (usually no longer than 255 characters [32]). This can be an excessive constraint for many practical applications in which the keys are typically long strings with many repeated parts. As far as the field of classical string-matching is concerned, there are many elegant data structures and powerful indices, such as compacted tries (in particular, suffix trees [1, 17, 22]) and suffix arrays [16, 20], which can handle unbounded-length texts and are characterized by very efficient searching and updating performance for small databases fitting into main memory. However, these tools are no longer efficient for the large databases that can change over time and that make considerable use of external memory. For example, suffix arrays [16, 20] make powerful searches possible by indexing *all* substrings but they cannot be changed any more easily than inverted files and they require *contiguous* space. In turn, suffix trees [1, 17, 22, 33] lose much of their efficiency in external memory because of the thrashing caused by their unbalanced tree topology (as we discussed in [13]). Nevertheless, since suffix trees are expected to achieve good average performance in external memory, they have been widely studied with the aim of finding a space-efficient and practical implementation for them. It has been shown that both suffix trees and compacted tries occupy $17N$ bytes when implemented carefully [16]. An improvement on this (i.e., $12N$ bytes) was achieved by Patricia trees [25], which are a space-efficient variant of compacted tries obtained by using the binary representation of each suffix and by storing only one bit per arc. PaTries [29], PAT-trees [16] and LC-tries [2] are variants of Patricia trees in which some heuristic is employed to further reduce the space to an average of $6N$ bytes. The most efficient implementation known is the Compact PAT-tree [8], which requires an average of $5N$ bytes and experimentally obtains a page fill ratio of at least 40–50%. Some other efficient implementations of suffix trees have also been investigated in a probabilistic setting by using some heuristics; their efficiency, however, depends on the text chosen (e.g., [12, 23]).

Notice that the applications considered so far are *static* in the sense that their text string set Δ is fixed and has been preprocessed once. However, if we allow this string set to change under the insertion and deletion of some individual strings (as required in Problem 1), the situation becomes more complex and the performance of all known methods degenerate dramatically [8]. In summary, all the well-known data structures for indexing large texts fail to solve Problem 1 efficiently.

We have recently proposed a data structure, which we called *SB-tree* [13], that combines the best of both B-trees and suffix arrays and that overcomes the limitations of inverted files (modifiability and atomic keys), suffix arrays (modifiability and contiguous space), suffix trees (unbalanced tree topology) and prefix B-trees (bounded length keys). The computational model we use to study the SB-tree performance is the classical *two-level memory* machine [11], which has a small fast main memory and a large slow external memory. This model assumes that the external memory is partitioned into *disk pages*, each of which contains B atomic items (e.g.,

[1] In many practical applications, however, we only need to search for the pattern occurrences in Δ's strings that start at fixed positions, called *index points* (e.g., the beginning of words).

integers, pointers, characters, etc.), where B is a parameter called *disk page size*. A single *disk access* transfers one disk page into main memory. The resulting algorithmic complexity consists of two main components: (1) the total number of disk accesses performed by the various operations, and (2) the total number of disk pages occupied by the data structures.

SB-trees are the first data structure for solving Problem 1 with provably good worst-case bounds in the two-level memory machine. These bounds are the following: Searching for all the *occ* occurrences of P in the strings of Δ takes $O(\frac{p+occ}{B}+\log_B N)$ worst-case disk accesses. Inserting in or deleting a string of length m from Δ takes $O(m \log_B(N+m))$ worst-case disk accesses. The space usage is $\Theta(\frac{N}{B})$ disk pages (see [13]).

Thanks to these favorable results, the SB-trees' theoretical behavior needs to be established experimentally. In this paper, we aim at describing an efficient SB-tree implementation on which we perform a large number of searching and updating experiments. We show that SB-trees are really very efficient in practice and thus can play an important role in the previously-mentioned applications. Furthermore, it is worth noting that SB-trees' applicability extends beyond indexing *dynamic* text collections because they are also competitive for searching on large *static* text collections. Our main findings are the following:

- We obtain a fast search that requires approximately $2h$ disk accesses, where $h = \Theta(\log_B N)$ is the SB-tree height. For example, we guarantee $h \leq 3$ for $N = 2$ billion indexed suffixes by using a page size of $B = 32$ kilobytes. The small height does not depend on the text content (i.e., its characters' distribution). This is not the case with suffix trees and compacted tries because their unbalanced tree topology is caused by their internal nodes that are in correspondence with some repeated substrings: We may therefore traverse a k-node path in them that occupies $\Omega(\frac{k}{\log_2 B})$ disk pages in the worst case. [2] The implementation of SB-trees is based on Patricia trees (which help us in routing SB-tree traversal) and their compressed representation (which reduces space and is also alphabet-independent). This allows us to achieve a large branching factor and a small SB-tree height. We do not only provide and experiment an SB-tree implementation occupying $12.25N$ bytes, but we also introduce and discuss a general technique that allows us to reduce the space usage to only slightly more than $4N$ bytes.
- We perform efficient updating by means of some algorithms that differ from the theoretically "good ones" in [13] in order to exploit the LRU buffering strategy imposed by the operating system better. The update algorithms in [13] treat one suffix at a time and avoid rescanning its characters. In this paper, we achieve a worse theoretical performance by treating batches of suffixes but we access disk pages in a *regular* way and therefore avoid making thrashing necessary. We show that updating SB-trees with this approach is five times faster than updating UNIX Prefix B-trees [32], and SB-trees require even less space. SB-tree updating inherits all of B-trees' advantages with respect to unbalanced trees in external memory: we control both the height and disk page fill ratio (which is guaranteed to be more than 90%) by using available technology for B-trees [5, 6, 10]. This makes SB-trees more appealing and effective than suffix trees and compacted tries when managing large external-memory text collections.

The rest of our paper is organized as follows. In Section 2, we summarize the basic ideas and properties underlying the design of SB-trees. In Section 3, we describe an efficient implementation method for SB-trees based on Patricia trees and succinct encodings that is: (a) *space-efficient* and (b) *alphabet-independent*. We then describe how to search for an arbitrary pattern in an SB-tree that uses this succinct representation and how to update SB-trees under single string insertions. In Section 4, we present a large number of searching and updating experiments that allow us to

[2] We noticed that, in some cases, there are some long, repeated substrings of about 10^4 characters — e.g., in manuals — which are caused by some cut-and-paste operations.

evaluate the practical performance of our SB-tree implementation. We show that SB-trees lead to much faster searches and can be updated in a reasonable amount of time. Finally, in Section 5, we list and discuss some topics that we believe require further investigation and experimentation.

2 The SB-tree data structure

We adopt standard terminology for a string $\delta[1,s]$ and call $\delta[1,i]$ a *prefix*, $\delta[j,s]$ a *suffix* and $\delta[i,j]$ a *substring* (for $1 \leq i \leq j \leq s$). We say that there is an *occurrence* of a *pattern* string P in δ if we can find a substring $\delta[i, i+|P|-1]$ equal to P.

Let us examine our Problem 1: We let $SUF(\Delta)$ be the set of all the suffixes of Δ's strings and number the suffixes in $SUF(\Delta) = \{S_1, S_2, \ldots, S_N\}$ in increasing lexicographic order, denoted by $\leq_L$. Given a pattern P, we say that P's *position* in $SUF(\Delta)$ is $j \geq 1$ if there are $j-1$ strings lexicographically smaller than P in $SUF(\Delta)$. From now on we assume that P's last character is smaller than any other character in the strings' alphabet and that each of Δ's strings is allocated in a contiguous segment of disk pages so that the page containing its i-th character can be located by a constant number of simple arithmetic operations on the string pointer.

The SB-tree design is based on Manber and Myers' crucial observation [20] that substring searching in Problem 1 can be divided into two steps: (1) we retrieve P's position in $SUF(\Delta)$'s strings in lexicographic order; (2) we list all of the pattern occurrences which are given by the *contiguous* sequence of strings following the pattern's position in $SUF(\Delta)$ and having the pattern as a prefix. From this observation, it follows that any external dynamic data structure solving Problem 1 should be designed with the aim of meeting three requirements; (a) it should occupy optimal space, i.e., $\Theta(N/B)$ disk pages; (b) perform efficient searching for a pattern's position in $SUF(\Delta)$; and (c) maintain set $SUF(\Delta)$ lexicographically ordered under string insertions and deletions.

SB-trees meet all three requirements above [13]. We now review the SB-trees' main features. From a high level point of view, they are B^+-trees (i.e., their keys reside in the leaves and their internal nodes only contain some copies of those keys [10]), where the keys are the *logical pointers* to $SUF(\Delta)$'s strings and the order between any two keys is the $\leq_L$-order among the corresponding pointed strings. We assume that each disk page can contain up to $2b$ keys in which the choice of $b = \Theta(B)$ depends on the disk page size B. In describing the logical organization of SB-trees, we adopt the convention that there is *no distinction* between a key and its corresponding pointed string.

The SB-tree SBT_Δ built on string set Δ is defined as follows: Each node π is stored in a disk page and contains an ordered string set $\mathcal{S}_\pi \subseteq SUF(\Delta)$, such that $b \leq |\mathcal{S}_\pi| \leq 2b$. We use $L(\pi)$ to denote the leftmost (resp., $R(\pi)$ the rightmost) string in $\mathcal{S}_\pi$. We distribute $SUF(\Delta)$'s strings among the SB-tree nodes as follows:

- We partition $SUF(\Delta)$ into groups of b strings except for the last group, which can contain from b to $2b$ strings. We map each group into a leaf π (and form its string set $\mathcal{S}_\pi$) in such a way that the left-to-right scanning of the SB-tree leaves gives us $SUF(\Delta)$. Furthermore, we associate the longest common prefix length $lcp(S_j, S_{j+1})$ with each pair S_j, S_{j+1} of $\mathcal{S}_\pi$'s adjacent strings.
- Each internal node π of SBT_Δ has $n(\pi)$ children $\sigma_1 \ldots \sigma_{n(\pi)}$, with $\frac{b}{2} \leq n(\pi) \leq b$ (except for the root, which has from 2 to b children). Set $\mathcal{S}_\pi$ is formed by copying the leftmost and the rightmost strings contained in π's children. That is, $\mathcal{S}_\pi$ is defined as the ordered string set $\{L(\sigma_1), R(\sigma_1), L(\sigma_2), R(\sigma_2), \ldots, L(\sigma_{n(\pi)}), R(\sigma_{n(\pi)})\}$ (see Figure 1).

Since the SB-tree's branching factor is $\Theta(b)$, its height $H(N,b)$ is such that $H(N,b) = O(\log_b N) = O(\log_B N)$. We refer the reader to Figure 2 for an example of an SB-tree in which

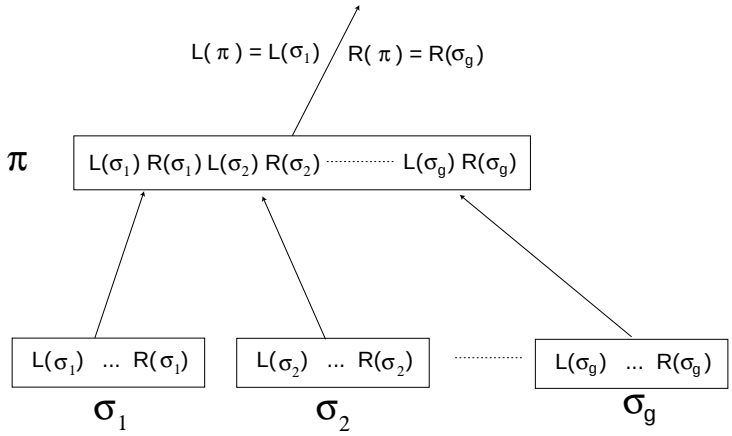

Figure 1: The logical organization of SB-tree internal nodes.

the keys are the logical pointers to $SUF(\Delta)$'s strings and we allocate Δ's strings in a contiguous segment of disk pages.

Since $SUF(\Delta)$ is stored in the SB-tree leaves and considering Manber and Myers' observation, we can conclude that pattern *searching* mainly consists of traversing SBT_Δ and retrieving P's position in $SUF(\Delta)$. In fact, given the SB-tree leaf containing that position, say $\widehat{\pi}$, all the *occ* occurrences are obtained by the *occ* suffixes in $SUF(\Delta)$ which have P as a prefix and are contained in the *contiguous* sequence of SB-tree leaves following $\widehat{\pi}$ (inclusive). The algorithmic scheme of the search operation is described in Figure 3. At this point, we introduce the following:

(a) List-occurrence(P,S) is a procedure that starts at suffix S, scans $SUF(\Delta)$ rightward, and outputs all the suffixes until one whose prefix *is not* P is encountered. We can implement List-occurrence by scanning the list of SB-tree leaves rightward starting at the leaf that contains S and checking if P is a prefix of a suffix in $SUF(\Delta)$ by means of the *lcp*-information contained in each SB-tree leaf (we must have $lcp \geq |P|$).

(b) Search-pos(P,$\mathcal{S}_\pi$) is a procedure that finds P's position in the string set $\mathcal{S}_\pi$. Since it is rather difficult to implement Search-pos(P,$\mathcal{S}_\pi$) efficiently, we postpone its discussion to Section 3. We wish to point out that we could search for the pattern's position by performing a binary search on $\mathcal{S}_\pi$'s strings and we would thus access at least $\log_2 b$ disk pages (at least one disk page for each string examined because it is implemented by its logical pointer). However, our approach consists of "properly" organizing set $\mathcal{S}_\pi$ by means of the *blind trie* data structure [13] and this allows us to reduce the number of strings examined from $\log_2 b$ to a single one !

We now discuss the pseudocode shown in Figure 3. In Steps (1)-(2), we check to see if $P \leq_L S_1$ or $P >_L S_N$, with $O(\frac{p}{B})$ disk accesses. In the former case, we execute List-occurrence(P,S_1); as far as the latter case goes, we are sure that P does not occur in Δ's strings. In all the other cases (i.e., $S_1 <_L P \leq_L S_N$), we traverse SBT_Δ and maintain the invariant: $L(\pi_i) <_L P \leq_L R(\pi_i)$ for each traversed node π_i (at level i, with $1 \leq i \leq H(N,b)$). The invariant holds for the SB-tree root π_1 because we ensured $S_1 < P \leq S_N$ with $S_1 = L(\pi_1)$ and $S_N = R(\pi_1)$. In the generic ith step, we route the SB-tree traversal by loading π_i's disk page (Step (4)) and find out the pattern's position j in $\mathcal{S}_{\pi_i}$ by means of Search-pos(P,$\mathcal{S}_{\pi_i}$) (Step (5)). In this way, we can determine two consecutive strings $X_{j-1}, X_j \in \mathcal{S}_{\pi_i}$, such that $X_{j-1} <_L P \leq_L X_j$. At this point, if π_i is a leaf (Step (6)), then we have found P's position in $SUF(\Delta)$ and we exit the while loop and list all of P's occurrences by means of List-occurrence(P,X_j) (Step (9)). If π_i is not a leaf, we have to maintain the induction by going deeper into SBT_Δ according to two other cases. In the first case

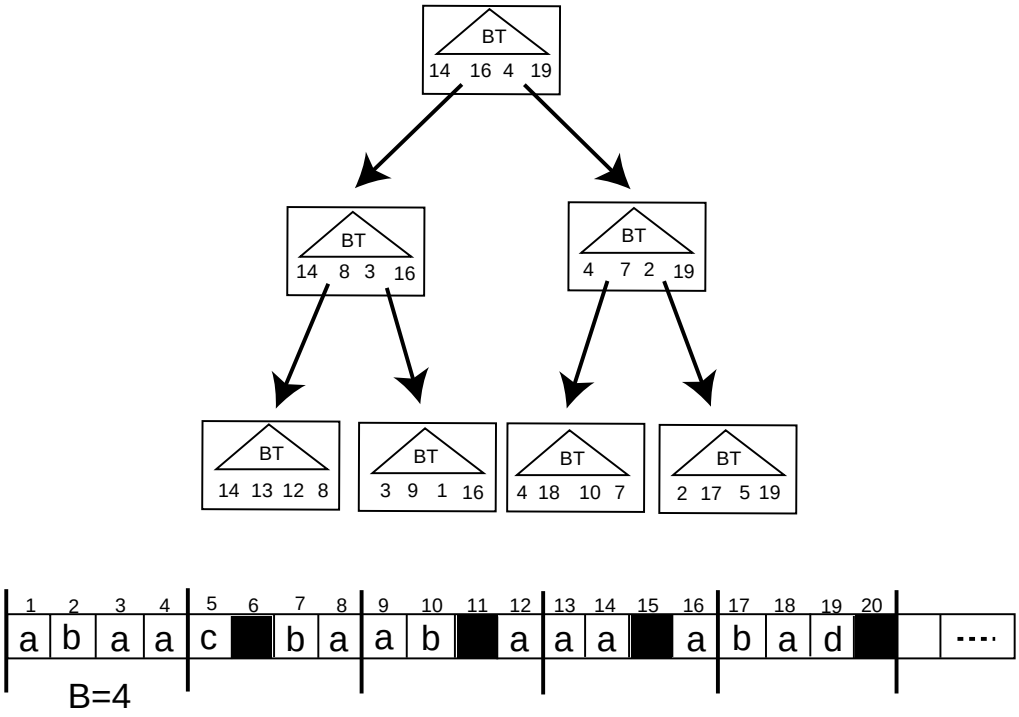

Figure 2: An example of SB-tree, where external memory is represented by a linear array with disk page size $B = 4$. The numbers in the SB-tree nodes are the logical pointers to $SUF(\Delta)$'s strings (i.e., their starting positions in external memory). The black boxes in the disk pages denote special endmarkers that prevent two suffixes of Δ's strings from being equal.

procedure *Searching for* P;

(1) **if** $P \leq_L S_1$ **then** *List-occurrence*(P,S_1); **return**;
(2) **if** $P >_L S_N$ **then return**;
(3) $\pi_1 :=$ *root*; $i := 1$;
while true do /* *Invariant:* $L(\pi_i) <_L P \leq_L R(\pi_i)$ */
(4) Load π_i's disk page and let $\mathcal{S}_{\pi_i} = \{X_1, X_2, \ldots, X_{2n(\pi_i)}\}$
(5) $j :=$ *Search-pos*(P,$\mathcal{S}_{\pi_i}$); /* $X_{j-1} <_L P \leq_L X_j$ */
(6) **if** π_i is an SB-tree leaf **then exit-while**;
(7) **if** $X_j = R(\sigma)$, for a child σ of π_i **then** $\pi_{i+1} := \sigma$; $i := i + 1$;
(8) **if** $X_j = L(\sigma)$, for a child σ of π_i **then** move to σ's leftmost descending leaf; **exit-while**;
endwhile
(9) *List-occurrence*(P,X_j).

Figure 3: The pseudocode for searching P in Δ's strings by means of SBT_Δ.

(Step (7)), we know that $X_{j-1} = L(\sigma)$ and $X_j = R(\sigma)$, for some child σ of π_i, and hence we set $\pi_{i+1} := \sigma$ to repeat the search in π_{i+1} (thus maintaining the induction). In the second case (Step (8)), we know that X_{j-1} and X_j belong to two distinct children of π_i so these two strings are *adjacent* in $SUF(\Delta)$ (because of the SB-tree's organization). Since $X_j = L(\sigma)$ for a child σ of π, we move to σ's leftmost descending leaf, exit the while-loop, and execute List-occurrence(P,X_j) in order to list all the pattern occurrences.

It is worth noting that the algorithmic structure of the pseudocode in Figure 3 is quite simple except for the blind trie implementation of Search-pos, which we describe in detail further on. As far as the complexity is concerned, it is possible to avoid rescanning the pattern's characters during the SB-tree traversal in order to achieve $O(\frac{p+occ}{B} + \log_B N)$ disk accesses (we refer the reader to [13]).

We do not go into the details of worst-case efficient SB-tree updates, as we discussed in [13], because we will present in Section 3.2 a simpler approach that is very efficient in practice.

3 A Practical Implementation of SB-Trees

We now describe an efficient SB-tree implementation, and this can be considered as the first serious step towards studying SB-trees' practical impact. From the above discussion, it follows that the SB-tree search's efficiency is strictly related to the effective blind trie implementation of Search-pos. In Section 3.1, we therefore examine this issue and provide a blind trie implementation based on Patricia trees and succinct encodings. The implementation is: (a) *space-efficient* and (b) *alphabet-independent*. These two characteristics are also important in practical applications because the case of a *large* alphabet is realistic: In computational linguistics, a text is often considered to be a list of *tokens* (i.e., pointers to words, q-grams, etc.) which forms the alphabet; therefore, the alphabet can have so many characters that is not possible to encode each of them by a single byte [7].

In Section 3.2, we explain how to search for an arbitrary pattern in an SB-tree by using the succinct blind trie representation. We then show how to update SB-trees under string insertions and provide an algorithm that takes advantage of the LRU buffering strategy imposed by the underlying operating system. (Since the deletion algorithm is simpler, it is not explicitly described.)

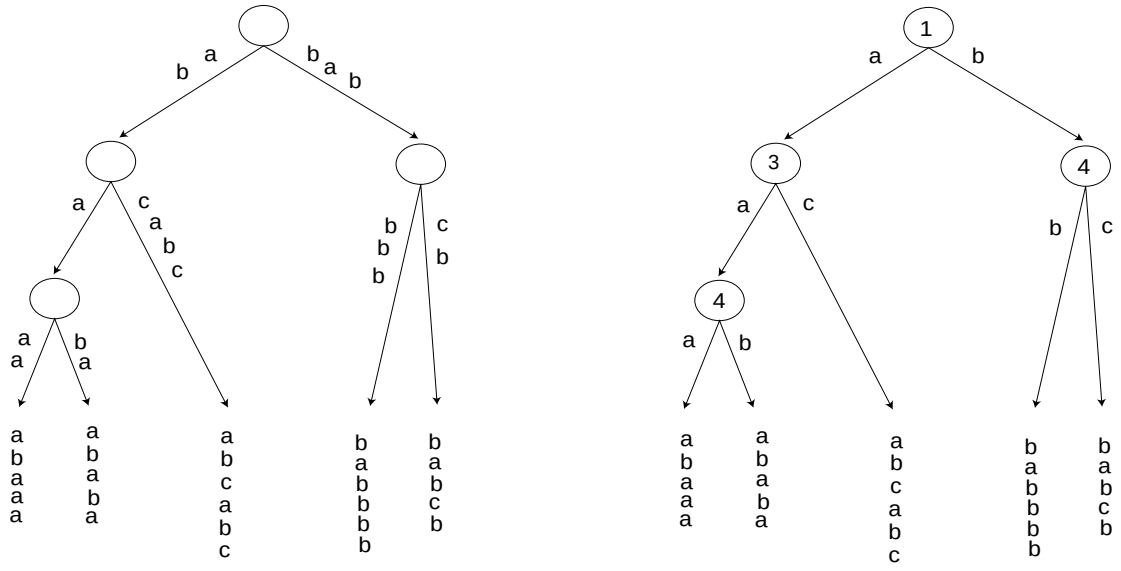

Figure 4: A compacted trie (left) and its corresponding blind trie (right).

3.1 Blind trie compression and uncompression

One of the main advantages of SB-trees is that their height $H(N, b)$ decreases exponentially as b's value increases (with fixed N), and this consequently has a favorable influence on the searching and updating performance. Value b is strictly related to the number of strings contained in each node π because $b \leq |\mathcal{S}_\pi| \leq 2b$. Therefore, we can speed up the SB-tree operations by increasing the number of strings able to be stuffed into $\mathcal{S}_\pi$ as much as possible. If the disk page size B increases, we can store more suffixes in $\mathcal{S}_\pi$. However, since B is limited by the typical size of a disk page (32 kilobytes) which is fixed a priori, we need a technique that maximizes $|\mathcal{S}_\pi|$ for a fixed B in order to squeeze as many strings as possible into one disk page. We keep this in mind in describing blind tries.

We define the blind trie BT_π plugged into an SB-tree node π in three steps (see Figure 4): (1) We build a compacted trie whose leaves store $\mathcal{S}_\pi$'s strings and whose arcs are labeled by their substrings. (2) We label each node u by integer $pos(u)$, which is equal to *one plus* the length of the string obtained by concatenating the arc labels in the downward path leading to u. (3) We delete all the characters in each arc label except the first one, which we refer to as the *branching character*. The total size of BT_π depends not only on $|\mathcal{S}_\pi|$ but also on the alphabet's size (because of the branching characters). Since each character in a large alphabet may require many bytes to be represented, a straightforward storage of blind tries would be alphabet-dependent and thus space-inefficient. Specifically, let us compute the amount of space required by the internal structure of BT_π (see Figure 4). $|\mathcal{S}_\pi|$ leaves store the pointers to the strings in $\mathcal{S}_\pi$, with no more than $|\mathcal{S}_\pi|-1$ internal nodes and $2|\mathcal{S}_\pi|-2$ arcs, and each node needs an integer for its *pos*-value and each arc requires a character and a branching pointer. If we assume that the pointers and integers occupy four bytes and each character is represented by at least one byte, a straightforward blind trie implementation would require a total space of at least $18|\mathcal{S}_\pi| - 14$ bytes, which is too large compared to the suffix tree implementations discussed in the introduction.

Blind trie space reduction is therefore the first problem we deal with in this section to make SB-trees really good in practice. We now show a space-efficient and alphabet-independent implementation of blind tries based on Patricia trees [19] that requires a total of $8.25|\mathcal{S}_\pi| - 4.25$ bytes

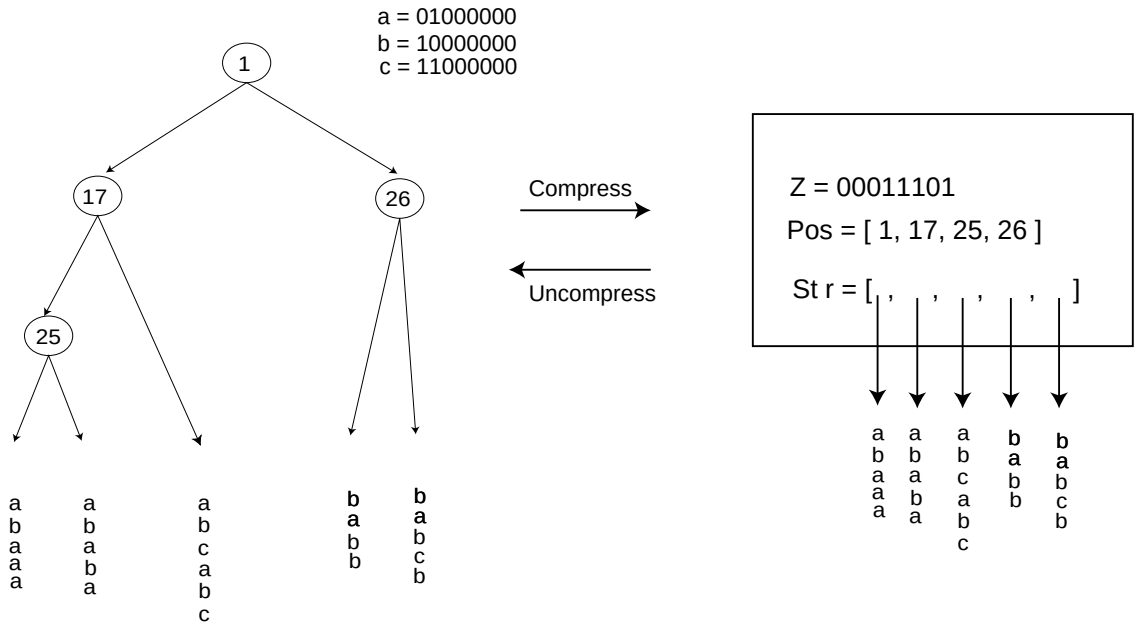

Figure 5: The Patricia Tree corresponding to the blind trie in Figure 4. The bit array Z, the array of *pos*-values, and the *Str*-array of string pointers are shown.

(see Figure 5, left). Let us examine $\mathcal{S}_\pi$'s strings in the form of binary sequences whose characters all have a fixed-length binary representation that maintains the $\leq_L$-order (we encode the string endmarkers by a *null* byte followed by a four-byte counter to make sure that $\mathcal{S}_\pi$'s strings are distinct, see Figure 2). The binary representation makes us sure that the internal node fan-out is exactly *two* in BT_π. Consequently, there is a total of $2|\mathcal{S}_\pi| - 1$ nodes. We express $pos(u)$ *in bits* for each internal blind trie node u because a *branching character* is now a single bit (hereafter called *branching bit*) and then we associate an *implicit binary* label to each arc (zero for each left arc and one for each right arc). We wish to point out that this blind trie representation is independent of the alphabet's size and therefore it matches the above requirements.

Although the blind tries' properties are very promising, we go a step further and use a simple method for representing BT_π *succinctly*. In this way, we save more space (i.e., we stuff more strings into $\mathcal{S}_\pi$) when storing BT_π in external memory. We exploit the property that each node of BT_π is either a leaf or an internal node having exactly two children and we design two operations: *Compress* and *Uncompress* (see Figure 5). We use the former for producing a succinct (and implicit) representation that can only be used for storing BT_π in external memory inside π's disk page. We use the latter for explicitly retrieving BT_π from its implicit (external-memory) representation when π is transferred to main memory in order to perform some computation on it. More precisely:

- *Compress(BT_π):* This procedure represents BT_π by means of three arrays: Pos, Str, Z, which are obtained by its preorder traversal (see Figure 5, right). The first two arrays require a total of $8|\mathcal{S}_\pi| - 4$ bytes. Array Pos contains the *pos*-value sequence found in the internal blind trie nodes. Array Str contains the string pointers sequence found in the blind trie leaves. Array Z is a binary array that encodes the blind trie shape and occupies a total of $\frac{|\mathcal{S}_\pi|-1}{4}$ bytes (only one bit per arc is used). [3] We build Z by traversing BT_π in preorder and by appending 0 for each

[3] Z's length is $2|\mathcal{S}_\pi| - 2$ bits but we can easily reduce it to $|\mathcal{S}_\pi| - 2$ because the arcs leading to the blind trie

left-branch and 1 for each right-branch. The total space used is $\frac{33}{4}|\mathcal{S}_\pi| - \frac{17}{4} = 8.25|\mathcal{S}_\pi| - 4.25$ bytes.

- *Uncompress(π):* This procedure allows us to reconstruct BT_π from the three arrays stored inside π's disk page (and generated by *Compress*). First of all, BT_π's shape is reconstructed by means of array Z as follows: Since each internal blind trie node has a fan-out of two, we scan Z rightward using a stack ST. Let x be the node at ST's top (at the beginning x is the blind trie's root), and assume that $i-1$ bits have already been scanned in Z. If $Z[i] = 0$, we create x's left child which we push onto ST's top only if $Z[i+1] = 0$ (otherwise, it is a leaf which we do not push onto ST). If $Z[i] = 1$, we create x's right child and pop x out of ST. If $Z[i+1] = 0$, then x's right child is an internal node which we push onto ST (otherwise, it is a leaf which we do not push onto ST). After reconstructing BT_π's shape, we set the *pos*-values contained in its trie nodes and the string pointers contained in its leaves by using the two arrays Pos and Str stored inside π's disk page.

We point out that π's disk page must also keep π's pointers to its SB-tree children because of the SB-tree's structure and this takes $4|\mathcal{S}_\pi|$ extra bytes. In brief, the total space needed for storing an SB-tree's node π (i.e., BT_π and the SB-tree child pointers) in external memory is $12.25|\mathcal{S}_\pi| - 4.25$ bytes. By making this quantity smaller than B (the disk page size), we make sure that an SB-tree node can be stuffed into only one disk page.

Lemma 1 *A node π of the SB-tree can be stored in $12.25|\mathcal{S}_\pi| - 4.25$ bytes, thus achieving a maximum branching factor $b \approx \frac{B}{24.5}$ for a disk page size B.*

Proof: Since $b \leq |\mathcal{S}_\pi| \leq 2b$, we only have to guarantee that $2b \leq B$. □

Let us now make some calculations to evaluate the SB-tree implementation better. For example, let us fix a disk page size $B = 32$ kilobytes (a typical disk track size). We fill up the disk pages as much as possible, and achieve a branching factor $b = 1336$. This allows us to store $N = 2$ billion suffixes in an SB-tree whose height is $H(N, b) = 3$ (by stuffing $2 \cdot 1336$ strings per SB-tree leaf). This occupies about $12.3N$ bytes in all. If we set $B = 1$ kilobytes instead, we obtain branching factor $b = 40$ and store $N = 2$ billion suffixes in an SB-tree whose height is $H(N, b) = 6$ (by stuffing $2 \cdot 40$ strings per SB-tree leaf). Clearly, these calculations do not depend on the text's input distribution but only on its size N. We remember that this is not true for suffix trees and compacted tries because their heights (and performance) strictly depend on the repeated substring lengths in the text's archive.

We believe that our SB-tree implementation deserves some further comments. If we compare SB-tree space usage to suffix tree one (i.e., an average of $5N$ bytes [8]), we can see that the former still seems too large. Nevertheless, the SB-tree structure is flexible enough to let us obtain an implementation taking a total of $(4 + \frac{4.25}{k})$ bytes (slightly more than the space usage of suffix arrays [16, 20]), where k is a proper positive parameter to be fixed. For this purpose, we only have to change the SB-tree leaves because the number of internal SB-tree nodes is negligible with respect to the number of its leaves (we have a large branching factor b). Since the children pointers in the SB-tree leaves are useless, we use their space to store some extra $SUF(\Delta)$'s strings. Moreover, we only build BT_π on the subset of $\mathcal{S}_\pi$'s strings formed by taking every other kth string. The total space taken up by the SB-tree leaves is as follows: (a) $8.25\frac{|\mathcal{S}_\pi|}{k} - 4.25$ bytes are needed for BT_π's compressed version which now only stores the $\frac{|\mathcal{S}_\pi|}{k}$ sampled strings; (b) $4(|\mathcal{S}_\pi| - \frac{|\mathcal{S}_\pi|}{k})$ bytes are needed to store the pointers to the rest of $\mathcal{S}_\pi$'s strings (i.e., the unsampled ones). Hence,

leaves do not need to be encoded. However, this does not yield significant space saving in the SB-tree.

procedure *Search-pos*(P,$\mathcal{S}_\pi$);

```
(1)   p_bit := p * char_code_len;
(2)   u := root of BT_π;
      /* First Phase */
(3)   while (u is not a leaf and pos(u) ≤ p_bit) do
(4)       if pos(u)-th bit of P is 0 then u := left_child(u) else u := right_child(u);
(5)   ℓ := one of u's descending leaves;
      /* Second Phase */
(6)   lcp := longest common prefix (in bits) of P and ℓ's string;
(7)   u := ℓ;
(8)   while (u ≠ root and pos(parent(u)) > lcp) do u := parent(u);
(9)   if (lcp + 1)st bit of P is 0
(10)      then j := # leaves to the left of u's leftmost descending leaf (exclusive);
(11)      else j := # leaves to the left of u's rightmost descending leaf (inclusive);
(12)  return j;
```

Figure 6: The pseudocode for finding P's position j in $\mathcal{S}_\pi$. We denote the number of bits encoding an alphabet's character by *char_code_len*.

this SB-tree organization requires $(4 + \frac{4.25}{k})|\mathcal{S}_\pi|$ bytes for an SB-tree leaf π and therefore a total of about $(4 + \frac{4.25}{k})N$ bytes for the entire SBT_Δ.

This change in the SB-tree's organization slightly affects the search operation. When we reach an SB-tree leaf, we can still be no more than k positions from our final destination. We therefore complete the operation by means of a binary search that examines $O(\log_2 k)$ candidate strings. We refer the reader to our conclusions regarding the time/space trade-off achievable by using this SB-tree organization.

3.2 Practical SB-tree searching and updating

In the following, we describe how to search and update SB-trees that use the previously mentioned succinct representation of blind tries.

Searching. We again examine an SB-tree's node π and its blind trie BT_π. Our aim is to design a Search-pos(P,$\mathcal{S}_\pi$) procedure that finds P's position in $\mathcal{S}_\pi$ efficiently and that benefits from the ideas on blind tries when implemented in the form of Patricia trees. We execute the pseudocode in Figure 6 for finding P's position j in $\mathcal{S}_\pi$ by examining *only one* string of $\mathcal{S}_\pi$ in the worst case.

Search-pos(P,$\mathcal{S}_\pi$) essentially consists of two main phases that make use of *the binary representation* of $\mathcal{S}_\pi$'s strings. In the first phase (Steps (3)–(5)), we trace a downward path from the root to a leaf in BT_π by only using *bit comparisons* to match the branching bit in each traversed arc with the corresponding P's bit. We stop the downward traversal either when we reach a leaf or we cannot further branch from the current node (in the latter case, we take an arbitrary descending leaf from that node). The leaf that we reach at the end of our traversal, say ℓ, may not necessarily give P's position in $\mathcal{S}_\pi$ but it has the nice property of *storing one of $\mathcal{S}_\pi$'s strings that shares its longest common prefix with P* (we can generalize the proof of Lemma 3 in [13] to Patricia trees by using a binary alphabet). We then begin the second phase (Steps (6)–(11)), in which we load ℓ's string and determine its longest prefix in common with P (let *lcp* be their longest common prefix length *in bits*). After that, we identify ℓ's ancestor u such that: either $u = root$ of BT_π or $pos(u) > lcp \geq pos(parent(u))$, where $parent(u)$ denotes u's parent in BT_π. All of the strings

stored in u's descending leaves share their first $pos(u) - 1$ bits and thus have P's first lcp bits as their prefix (no other string in $\mathcal{S}_\pi$ has this property). We therefore find P's position in $\mathcal{S}_\pi$ by moving either to the left of u's leftmost descending leaf (if P's $(lcp+1)$th bit is 0) or to the right of u's rightmost descending leaf (if P's $(lcp+1)$th bit is 1). The binary representation of $\mathcal{S}_\pi$'s strings notably simplifies case analysis and speeds up computation thanks to the fast *bit* operations. It is clear from the above description that the name "blind trie" derives from the way we perform the traversal: we only examine the branching characters and ignore the rest of the substrings (now implicitly) associated with the blind trie arcs.

Let us take Figure 5 for example: we examine pattern P = aac = 01000000 01000000 11000000 (where the characters are encoded as shown in the figure). The first downward phase determines leaf ℓ– the third one in $\mathcal{S}_\pi$. We do not actually find the pattern's position but we know that ℓ's string, i.e. abcabc, is one of $\mathcal{S}_\pi$'s strings that shares its longest common prefix with P (i.e., they share 8 bits, namely 01000000). We use the mismatch bit 0, i.e., the 9th bit in P, to retrieve the pattern's position in $\mathcal{S}_\pi$ (i.e., the leftmost one).

We only perform the disk accesses needed for computing lcp because P and the branching bits are available in main memory together with BT_π. However, the blind tries themselves do not guarantee the bounds we claim.

Consequently, we extend the SB-tree search illustrated in Figure 3 as follows: During the SB-tree traversal and for each visited node π_i, we maintain the *invariant* that there is at least one string in $\mathcal{S}_{\pi_i}$ that shares its first s_i characters with P, for a proper positive value s_i. This implies that we can only examine the characters in $P[s_i + 1, p]$ when comparing P to ℓ's string in Step (6) of Search-pos(P,$\mathcal{S}_\pi$) (Figure 6) because of ℓ's string property that $lcp \geq s_i \cdot char\_code\_len$. We therefore design a new Search-pos procedure that takes three input parameters P, $\mathcal{S}_\pi$ and s, where the new input parameter s denotes the number of characters shared by P and one of $\mathcal{S}_\pi$'s strings. The procedure now returns the pair (j, lcp_char), where j is P's position in $\mathcal{S}_\pi$ (as before) and the new output parameter lcp_char is the number of P's characters matched during the blind search (we readily have $lcp\_char = \left\lfloor \frac{lcp}{char\_code\_len} \right\rfloor$). From the above considerations, it follows that the new pseudocode for Search-pos(P,$\mathcal{S}_\pi$,s) can be obtained by only changing Step (6) in the pseudocode shown in Figure 6 so that P is compared to ℓ's string by starting from the $(s * char\_code\_len + 1)$-th bit. As a result, Search-pos(P,$\mathcal{S}_\pi$,s) takes $\left\lceil \frac{lcp\_char - s}{B} \right\rceil + 1$ disk accesses.

At this point, we also have to change the SB-tree search's pseudocode in Figure 3. We add instruction $s_1 := 0$ to line (3) and replace line (5) with:

$$(5)\ (j, s_{i+1}) :=\ \text{Search-pos}(P,\mathcal{S}_{\pi_i},s_i);$$

The correctness of the new SB-tree search's pseudocode readily follows. Let us assume that s_i satisfies the invariant, that is, at least one string in $\mathcal{S}_{\pi_i}$ shares its first s_i characters with P (the base case for $s_1 = 0$ trivially holds). Search-pos(P,$\mathcal{S}_{\pi_i}$,s_i) is correctly executed on π_i and then s_{i+1} is set to be the longest common prefix length of P and any string in $\mathcal{S}_{\pi_i}$ because of ℓ's string property (i.e., $s_{i+1} \geq s_i$). The invariant on s_{i+1} is therefore preserved because either $L(\pi_{i+1})$ or $R(\pi_{i+1})$ shares the first s_{i+1} characters with P since they are adjacent in $\mathcal{S}_{\pi_i}$ and $L(\pi_{i+1}) \leq_L P <_L R(\pi_{i+1})$ (see Figure 1).

As far as the complexity of the pseudocode in Figure 3 is concerned: Step (4) takes one disk access to retrieve π_i's page and Step (5) takes $\lceil \frac{s_{i+1} - s_i}{B} \rceil + 1$ disk accesses (see above). Consequently, the total cost of the SB-tree search is a telescopic sum $\sum_{i=1}^{H(N,b)} (\lceil \frac{s_{i+1} - s_i}{B} \rceil + 2)$ (where $s_1 = 0$ and $s_{H(N,b)} \leq p$) and we conclude:

procedure Insert(i,j,π):

(1) Load π's page and let $\mathcal{S}_\pi = \{X_1, \ldots, X_{2n(\pi)}\}$;
(2) **For** $r := i$ **to** j **do** $(pos[r], lcp[r]) :=$ Search-pos(Y_r,$\mathcal{S}_\pi$, $lcp[r]$);
(3) **For** $h := 1$ **to** $2n(\pi)$ **do** $\mathcal{K}_h := \{Y_r : pos[r] = h,\ i \leq r \leq j\}$;
(4) $\mathcal{K} := \mathcal{K}_1 \cup \{X_1\} \cup \mathcal{K}_2 \cup \{X_2\} \cup \cdots \cup \mathcal{K}_{2n(\pi)} \cup \{X_{2n(\pi)}\}$;
(5) **if** π is an SB-tree leaf **then**
(6) Partition $\mathcal{K}$ into t sets $\mathcal{K}'_1, \mathcal{K}'_2, \cdots, \mathcal{K}'_t$ of size between b and $2b$ each;
(7) **For** $h := 1$ **to** t **do** build a new node $\widehat{\pi}_h$ and set $\mathcal{S}_{\widehat{\pi}_h} := \mathcal{K}'_h$;
else /* π *is an internal SB-tree node with children* $\sigma_1, \ldots, \sigma_{n(\pi)}$ */
(8) **For** $h := 1$ **to** $n(\pi)$ **do**
(9) **If** $\mathcal{K}_{2h-1} \cup \mathcal{K}_{2h}$ is empty **then**
(10) $\mathcal{L}_h := \{X_{2h-1}, X_{2h}\}$;
(11) **else** $\mathcal{L}_h :=$ Insert(f,g,σ_h) /* *where* $\mathcal{K}_{2h-1} \cup \mathcal{K}_{2h} = \{Y_f, \ldots, Y_g\}$ */
(12) Build new $\mathcal{S}_\pi := \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_{n(\pi)}$ and update the children pointers accordingly;
(13) Partition $\mathcal{S}_\pi$ into t sets $\mathcal{S}'_1, \mathcal{S}'_2, \ldots \mathcal{S}'_t$ of size between b and $2b$ each;
(14) **For** $h := 1$ **to** t **do** build a new node $\widehat{\pi}_h$ and set $\mathcal{S}_{\widehat{\pi}_h} := \mathcal{S}'_h$;
endif
(15) Discard π;
(16) **Return** list $\{L(\widehat{\pi}_1), R(\widehat{\pi}_1), \ldots, L(\widehat{\pi}_t), R(\widehat{\pi}_t)\}$.

Figure 7: The pseudocode for inserting $Y_i, \ldots, Y_j$ in the subtree rooted at π. $pos[r]$ is Y_r's position in $\mathcal{S}_\pi$ and $lcp[r]$ is the number of Y_r's characters matched.

Theorem 2 *We can search pattern $P[1,p]$ in SBT_Δ with no more than $\lfloor \frac{p}{B} \rfloor + 3\,H(N,b)$ worst-case disk accesses. We take no more than $\lceil \frac{occ}{b} \rceil$ disk accesses to retrieve all of the occ pattern occurrences in Δ's strings.*

We have proved experimentally that the searching cost is $2H(N,b)$, as shown in Table 1 (typically, $p << B$).

Updating. Although the insertion algorithm provided in [13] is very attractive in that it minimizes the number of disk accesses and avoids string rescanning (see the introduction), we now propose another algorithm that is theoretically less efficient but actually very fast in practice because we use two effective heuristics with it: (a) we perform a *batched* suffix insertion by keeping and sorting the suffixes of the string to be inserted in main memory before updating the SB-tree; (b) we exploit the LRU buffering strategy imposed by the underlying operating system by retrieving in preorder the SB-tree nodes involved in the updating process.

We let $Y[1:m]$ be the string to be inserted and we number its suffixes in lexicographic order: $Y_1, \ldots, Y_m$. The insertion of each suffix determines a root-to-leaf path traversal in the SB-tree. Since $Y_1, \ldots, Y_m$ are sorted, these paths are traversed in preorder and make efficient buffering of their disk pages possible. Our batch insertion procedure exploits this ordering by means of the following heuristics (not guaranteed by the procedure in [13]): each disk page storing a traversed SB-tree node is read and written only once even though many suffixes can be routed through it.

The overall insertion procedure consists of inserting all of Y's suffixes in batches of size s (we assume without any loss in generality that s divides m) by executing the procedure Insert($hs+1$,$(h+1)s$,root) for $h = 0, 1, \ldots, \frac{m}{s}$, whose pseudocode is given in Figure 7. Procedure Insert(i,j,π) is defined recursively by inserting the ordered sequence $Y_i, \ldots, Y_j$ (batch) in the subtree rooted at the SB-tree node π. The procedure returns the list formed by all the $L(\cdot)$'s and $R(\cdot)$'s strings

of the nodes that are possibly created by π's splitting because of the suffix insertion (Step (16)). This list is run through the recursive calls in order to maintain the SB-tree structure properly. In particular, we use the list returned by $\mathsf{Insert}(i,j,\pi)$ to update π's parent (if it exists) and its children pointers. We choose the batch size $s \leq b$ to guarantee that the recursive insertion creates one new node per node traversed at most. If the execution of $\mathsf{Insert}(hs+1,(h+1)s,\mathsf{root})$ returns more than two suffixes, the current SB-tree root has been split and so we build a new root (and increment the SB-tree's height by one).

We now wish to make some comments on the pseudocode in Figure 7 because it is fundamental in SB-tree updating. We use two global arrays $pos[1..m]$ and $lcp[1..m]$ in main memory, such that $pos[r]$ contains Y_r's position in the string set $\mathcal{S}_\pi$ of the current node π and $lcp[r]$ is the number of Y_r's characters matched so far. We let $\mathcal{S}_\pi = \{X_1, \ldots, X_{2n(\pi)}\}$ be the set of strings in $\mathcal{S}_\pi$. We merge $Y_i, \ldots, Y_j$ and $\mathcal{S}_\pi$'s strings together by performing a cumulative blind search: we carry $\mathsf{Search\text{-}pos}(Y_r,\mathcal{S}_\pi,\ lcp[r])$ out for all $i \leq r \leq j$ in Step (2) in order to exploit the SB-tree organization and we collect the suffixes having the same position in $\mathcal{S}_\pi$ (see Steps (3)). We do this to reduce the number of disk accesses with respect to a brute-force merging method. Let $\mathcal{K} = \mathcal{K}_1 \cup \{X_1\} \cup \mathcal{K}_2 \cup \{X_2\} \cdots \cup \{X_{2n(\pi)}\}$ denote the resulting merged sequence, where $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_{2n(\pi)}$ is a partition of $\{Y_i, \ldots, Y_j\}$ (Steps (4)). Two cases may now occur (Step (5)):

1. π is a leaf (Steps (6)–(7)). We partition $\mathcal{K}$ into some parts, say $\mathcal{K}'_h$, whose size ranges from b to $2b$ and we build a new node, say $\widehat{\pi}_h$, for each of them. These nodes substitute π in the SB-tree. For each $\mathcal{K}'_h$, we return the pair given by its leftmost and rightmost strings (i.e., $L(\widehat{\pi}_h)$ and $R(\widehat{\pi}_h)$). The whole list of pairs is returned in Step (16) to take into account π's possible splitting (because of the insertion of $Y_i, \ldots, Y_j$) and thus maintain the correctness of the recursive calls.

2. π is an internal node (Steps (8)–(14)). Due to the recursive insertion of $Y_i, \ldots, Y_j$ into π, its set $\mathcal{S}_\pi$ may need to be changed in order to take π's new children into account. We examine all of π's current children $\sigma_1, \ldots, \sigma_{n(\pi)}$ and perform a recursive insertion into each of them (if necessary, Step (9)). Let σ_h be the current h-th child, where $1 \leq h \leq n(\pi)$. If $\mathcal{K}_{2h-1} \cup \mathcal{K}_{2h}$ is empty, then σ_h is not modified and thus we reflect this by setting $\mathcal{L}_h = \{X_{2h-1}, X_{2h}\}$ (Step (10)). Otherwise, set $\mathcal{K}_{2h-1} \cup \mathcal{K}_{2h}$ contains some suffixes, say $Y_f, \ldots, Y_g$ with $i \leq f \leq g \leq j$, and thus we continue the recursive insertion of $Y_f, \ldots, Y_g$ into the subtree rooted at σ_h (Step (11)). This recursive insertion into a child σ_h may determine its substitution by some new nodes. In this case, we have to update $\mathcal{S}_\pi$ by substituting the string pair X_{2h-1} and X_{2h} (previously copied from σ_h into π due to the SB-tree's structure, see Figure 1) with the strings $L(\cdot)$ and $R(\cdot)$ belonging to the nodes created by inserting $Y_f, \ldots, Y_g$ into σ_h recursively. These strings are returned by the recursive insertion in σ_h and are stored in list $\mathcal{L}_h$ (Step (11)). Consequently, new set $\mathcal{S}_\pi := \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_{n(\pi)}$ correctly reflects these changes on π's children (Step (12)). After the recursive insertion, we proceed as in Case 1 where the new $\mathcal{S}_\pi$ plays the role of $\mathcal{K}$ and we may have to split $\mathcal{S}_\pi$ if it gets too large and create some new nodes (Steps (13) and (14)). We return the list of $L(\cdot)$'s and $R(\cdot)$'s strings for these nodes in Step (16) and thus maintain the correctness of the recursive calls.

Theorem 3 *Let lcp_i be the number of Y_i's characters matched during its insertion, $1 \leq i \leq m$. We take no more than $5\,m\,H(N+m,b) + \sum_{i=1}^{m} \lfloor \frac{lcp_i}{B} \rfloor$ worst-case disk accesses to insert all of Y's suffixes, where $H(N+m,b)$ is the height of the resulting SB-tree.*

Proof: By Theorem 2, Y_i's insertion requires no more than $\lfloor \frac{lcp_i}{B} \rfloor + 3\,H(N+m,b)$ disk accesses. In addition, there are no more than $H(N+m,b)$ page splits for each path leading to an updated

leaf and this costs no more than $2\,H(N+m,b)$ disk accesses because of the batch size $s \leq b$. The bound follows by summing up for $i = 1, \ldots, m$. $\square$

Although the $O(m\,H(N+m,b))$ worst-case bound stated in [13] does not contain the additional term $\sum_{i=1}^{m} \lfloor \frac{lcp_i}{B} \rfloor$ due to Y's rescanning, we discovered that $\frac{lcp_i}{B} < 1$ (i.e., we rarely exceeded the disk page size in making string comparisons) and the number of page faults was actually much lower than $2\,m\,H(N+m,b) + m$. Furthermore, the number of updated disk pages was strictly lower than m (see Table 2), and so the theoretical bound we stated in [13] is pessimistic because our insertion procedure does not usually create many problems in practice.

4 Experimental Results

Our experiments confirmed our theoretical predictions regarding SB-trees: (a) they lead to much faster searches (because $\log_b N$ is much smaller than $\log_2 N$); (b) they can be updated in a reasonable amount of time.

We compared SB-trees to a suffix array implementation that we developed in C-language and a Prefix-B-Tree implementation [6] available in the Unix operating system [32]. Unfortunately, no external-memory suffix tree implementation was available and we did not develop one by ourselves because we did not want to run the risk of underestimating the suffix tree's real performance. As a matter of fact, recent heuristics [2, 8] for managing suffix trees in external memory are quite complex and fully of latent details. We therefore decided to use the best-known experimental results currently available in open literature [8] as our criteria in comparing SB-trees to suffix trees.

We ran the experiments on varying amounts of two entirely different kinds of real-world texts: (1) an Associated Press newswire (henceforth, AP text); (2) a database of telephone numbers and billing addresses (henceforth, AT&T phone book). The text sizes ranged from 1 to 128 megabytes and we fixed a disk page size of $B = 32$ kilobytes (unless otherwise specified) because it is the standard magnetic disk track size. We used a Sun Sparc IPX with 32 megabytes of main memory and 2 gigabytes of external memory.

4.1 Searching experiments

We begin the evaluation of the SB-trees' searching performance by comparing it with the one we designed for suffix arrays.

Our implementation of the suffix array search is facilitated by our being able to store both the suffix array and the text in disk pages. This allows us to retrieve $\frac{B}{4} = 8192$ suffix pointers or $B = 32768$ text characters by means of a single disk access. The search consists of a binary search on the *set of disk pages* storing the suffix array by only using the leftmost and rightmost suffix of the page we visit. As a result, the approach saves the last $\log_2 \frac{B}{4}$ lookups on the suffix array because they are all on the same page.

The search performance of both approaches was evaluated by counting the total number of disk accesses empirically. We considered these accesses to be the sum of the accesses to the disk pages that stored both the indexing data structure (either suffix arrays or SB-trees) and the text (due to $\leq_L$-comparisons). Specifically, we counted a *new disk access* whenever the accessed disk page of the indexing data structure was different from the previous one (both on suffix arrays and SB-trees). We counted only *one* disk access per $\leq_L$-comparison when dealing with suffix arrays (even though it might take more than one in practice), while we counted the *exact* number of accesses per $\leq_L$-comparison when dealing with SB-trees. As a result, our evaluation of the page

N	Suffix Array		SB-Tree		$H(N,b)$
	Ave	Max	Ave	Max	
1	$30.770 \pm .176$	38	$3.846 \pm .021$	4	2
2	$34.244 \pm .187$	41	$3.964 \pm .010$	4	2
4	$37.325 \pm .166$	44	$5.985 \pm .006$	6	3
8	$39.861 \pm .177$	47	$5.994 \pm .004$	6	3
16	$42.474 \pm .180$	50	$5.991 \pm .004$	6	3
32	$44.913 \pm .172$	52	$5.996 \pm .005$	6	3
64	$46.948 \pm .176$	55	$5.972 \pm .007$	6	3
128	$49.797 \pm .182$	57	$5.993 \pm .006$	6	3

Table 1: The performance of searching a pattern of length 16 on AP-news. N = millions of indexed suffixes; Ave = average number $\pm$ standard deviation of disk accesses; Max = maximum number of disk accesses; $H(N,b)$ = height of the SB-tree.

faults is *optimistic* for suffix arrays and *pessimistic* for SB-trees. Although the simplifications we introduced are somewhat conservative, our results are definitely independent of the buffering strategy used by the underlying operating system and allowed us to deal with the searching performance both theoretically and experimentally, without having to introduce a lot of other variables (such as the main memory size, the cache size, the buffering strategy used by the operating system, etc.).

The search performance of both approaches was evaluated by counting the total number of disk accesses as the sum of accesses to the disk pages storing both the indexing data structure (either suffix arrays or SB-trees) and the text (due to $\leq_L$-comparisons). We measured the total number of disk accesses empirically by counting a *new disk access* whenever the accessed disk page was different from the previous one. This gave us an upper bound to the total number of page faults executed by a common operating system according to any caching strategy. Although this simplification is somewhat conservative, it made possible for us to deal with the searching and updating performance both theoretically and experimentally, without having to introduce a lot of other variables (such as the main memory size, the cache size, the buffering strategy used by the operating system, etc.). Our evaluation of the page faults is *optimistic* for suffix arrays and *pessimistic* for SB-trees. Specifically, when dealing with suffix arrays, we counted only *one* disk access per $\leq_L$-comparison, even though it might take more than one in practice. When dealing with SB-trees, we counted the number of disk accesses to the text pages *exactly*. Therefore, our results are definitely independent of the buffering strategy used by the underlying operating system.

In our experiments, we searched for some different pattern lengths p (ranging from $p = 2$ to $p = 128$ characters). Table 1 summarizes the results for AP texts with $p = 16$, and shows that SB-trees can be searched much more quickly than suffix arrays, presumably because the former contain a $\log_b$ term instead of a $\log_2$ term ($b >> 2$ occurs experimentally). The maximum number of disk accesses ranges from 4 to 6 for SB-trees and from 38 to 57 for suffix arrays. There is a gap in the SB-tree Ave-column of Table 1 when N increases from 2 megabytes to 4 megabytes because height $H(N,b)$ ranges from 2 to 3. The large set of searching experiments in Figure 8 clearly shows us that SB-trees achieved a 10 speed-up factor over suffix array performance.

Table 1 and Figure 8 also show that our search requires about $2\,H(N,b)$ disk accesses, where

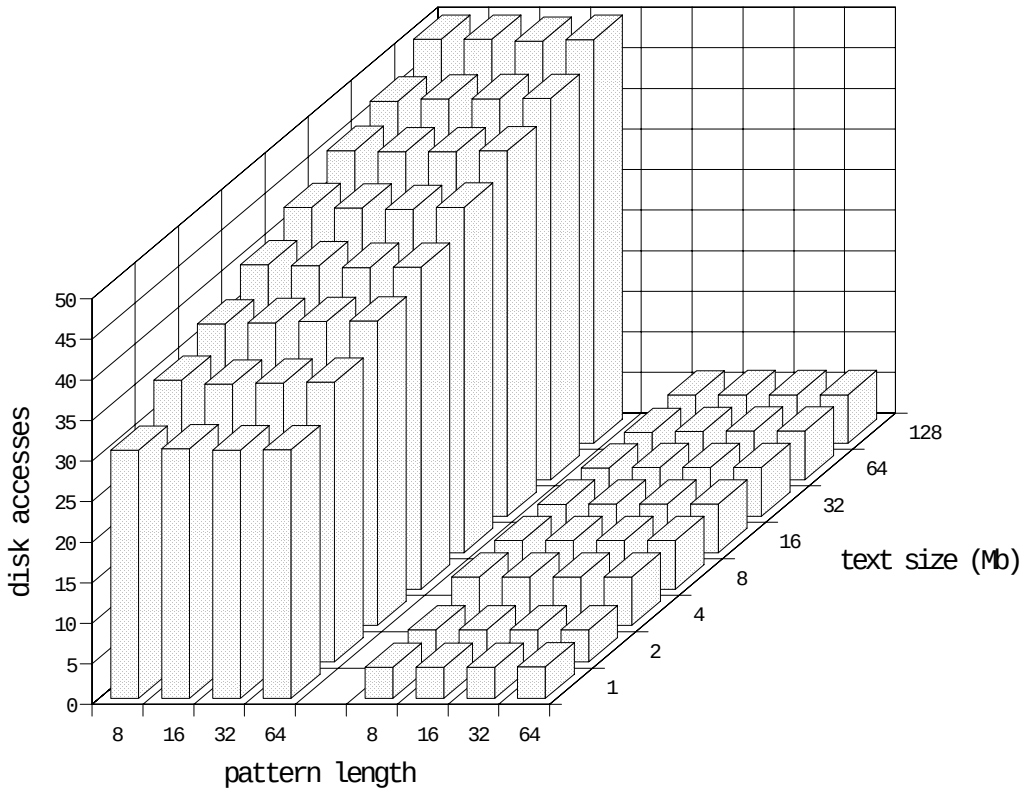

Figure 8: A summary of the whole set of our searching experiments. The left columns refer to suffix arrays and the right ones to SB-trees.

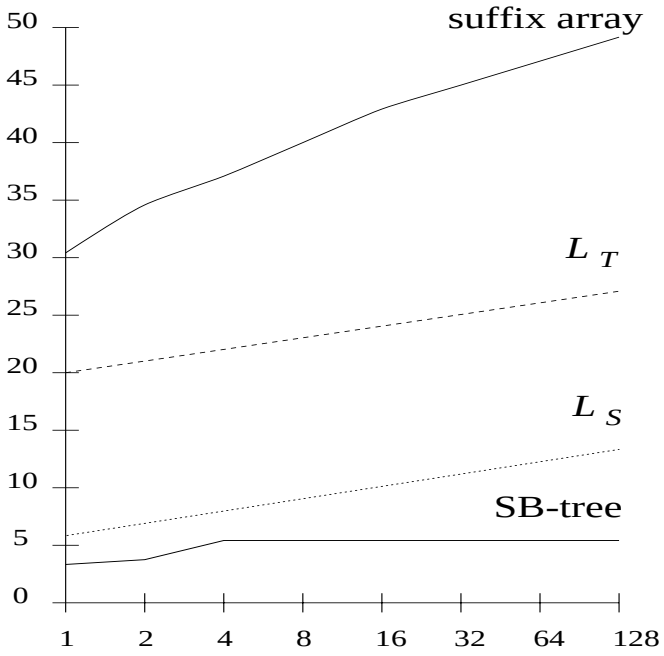

Figure 9: The average number of disk accesses (y-axis) per million indexed suffixes (x-axis).

$3\,H(N,b)$ is the worst-case bound stated in Theorem 2. Since we can fit $2b = 2672$ suffixes into each node of the SB-tree (see Lemma 1), $H(N,b)$ turns out to be very low and this influences the overall searching performance. We could conjecture that six disk accesses is the searching cost for 2 billion suffixes.

At this point, the reader may object that a smart implementation of the suffix array (binary) search could be faster than our search. However, if we use the average-case lower bound shown in [19], we obtain the minimum average number of disk accesses that any binary search in external memory must perform on the text, $L_T = \log_2 N$, and on the suffix array, $L_S = \log_2 \frac{N}{(B/4)}$ (see Figure 9). It is worth noting that $L_T + L_S$ is obviously smaller than the experimental number of disk accesses for suffix arrays but is larger than the one for SB-trees. This means that every implementation of the binary search on suffix arrays that does not make any empirical assumptions about the text character's distribution (e.g., an interpolation search [19]) can never overcome our SB-trees' practical performance.

At this point, some considerations are in order about our experimental results. One might observe the fact that suffix arrays' performance is worse than SB-trees' performance is not a real problem because current technology allows for a disk access time of about 10 milliseconds. Therefore, each suffix array search actually takes less than a second and this is definitely a reasonable waiting time for a human being. It would seem that SB-trees are a powerful theoretical tool that turns out to be relatively uninfluential in real environments. We feel that this is not true and support our opinion by the following example regarding network servers (seen in the light of the growing interest in networks and searching facilities, such as `http://inktomi.berkeley.edu`). A consequence of the access time reduction is that a server implementing SB-trees can provide a bandwidth at least 10 times larger than the one achievable with suffix arrays. Hence, it can accommodate more users in the same time interval. This makes a great difference in designing powerful searching facilities in distributed environments (e.g., project Dienst [14, pag. 47]). Furthermore, we should not forget that SB-trees' performance in processing text update operations

greatly surpasses all known methods (see the next section) and therefore makes SB-trees a valid (potential) tool for practical applications.

The last subject we deal with in this section is suffix tree implementation. As previously mentioned, we do not have an efficient suffix tree implementation in external memory and want to compare suffix trees to SB-trees in a conservative way. We take Clark and Munro's [8] best experimental results on suffix trees in external memory as our basis. The authors choose the Oxford English Dictionary [26] (OED) as a text archive of size $N \approx 128$ million indexed suffixes (actually its total size is half a gigabyte, but they only index word beginnings). They fix a page size of $B = 8$ kilobytes and find that the suffix tree height is 4 (the root is kept in main memory). This interesting result deserves a comment. The suffix tree's height and size *strictly* depend on the text characters' distribution and the lengths of some "repeated substrings" (because the internal suffix tree nodes store some substrings that occurs at least twice in the text). Moreover, the unbalanced suffix tree topology and the heuristics used to bucket their nodes in external memory cannot obtain more than a 43% disk-page fill ratio [8]. Conversely, SB-trees have a *guaranteed* performance that only depends on the text archive size N. Furthermore, SB-trees can achieve a 100% disk-page fill ratio by using current B-tree technology. This makes SB-tree performance very predictable, even for unknown text archives. Consequently, considering the effects of blind trie compression and uncompression and Lemma 1, it is very likely that the SB-tree is not higher than the suffix tree on the OED archive. Furthermore, as we discussed in Section 3.1, the balanced structure of SB-trees can be easily changed to require a space of slightly more than 4 bytes per indexed suffix and this is an improvement over suffix trees. It is also worth noting that SB-tree properties also hold in a *dynamic* setting where, instead, the use of suffix trees is very expensive because we cannot be sure that the latter are always balanced.

4.2 Updating experiments

Our experiments showed that SB-trees can be updated in a reasonable amount of time and they are at least as good as B-trees in this respect, whereas, in many other respects, they are definitely much better (e.g., in searching time and space saving). In particular, we compared their updating performance to: (1) an efficient implementation of the Prefix-B-Trees available in the Unix operating system [32]; (2) a "start-over" method that builds the suffix arrays from scratch by merging their previously sorted parts.

We studied the problem of updating an ordered set of N suffixes under the insertion of m other suffixes drawn from a given string Y, for several values of m and N. This problem arises in many practical applications in which significant pattern occurrences start at some positions in Y (e.g., at the beginning of words, paragraphs, etc.). We therefore let SUF_m be the set of these suffixes and took them in lexicographical order.

The first question was to find out how an SB-tree insertion behaves. We observed that, even though m was fixed (i.e., $m = 1$ million) and $N = 1, 2, \ldots, 32$ millions, the insertion time increased *linearly* with the number $\frac{N}{b}$ of SB-tree leaves ! (See Figure 10.) The reason was that $m \geq \frac{N}{b}$, and this caused the updating of almost *all the SB-tree leaves* because each leaf was expected to contain at least one new suffix from SUF_m. This basically meant rebuilding the data structure from scratch ! This phenomenon is strictly related to the B-tree structure of SB-trees and it is not very surprising because high m values often appear in strings, while they rarely seem to occur in B-trees with ordinary integer keys.

Keeping this in mind, we performed the same experiment with a smaller $m = 10^4$ but ran into the same problem (because we still had $m \approx \frac{N}{b}$). We decided to change the disk page size B from 32 kilobytes to 1 kilobyte (to increase the total number of SB-tree nodes). The number

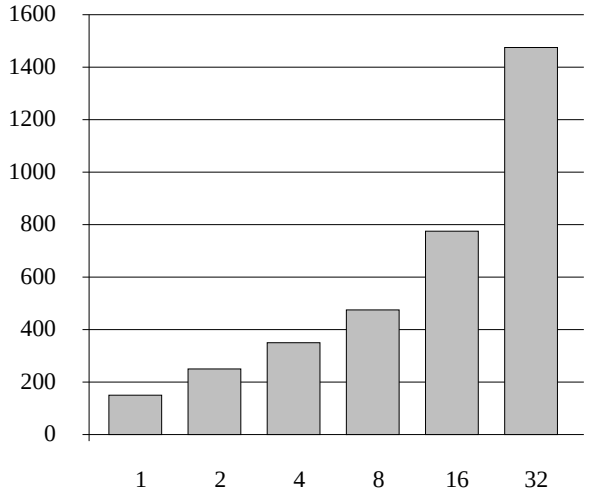

Figure 10: Inserting $m = 1$ million suffixes in a text archive of $N = 1, 2, \ldots, 32$ million suffixes.

#SUF	% updated leaves	# total leaves
1	58%	13108
2	18%	26215
4	12%	52429
8	8%	104858
16	3.5%	209716
32	1.5%	419431

Table 2: The insertion of $m = 10^4$ suffixes in #SUF million suffixes.

of SB-tree leaves ranged from about 13 thousand to 420 thousand and so $m < \frac{N}{b}$. A new set of experiments showed that the number of updated pages was no longer proportional to $\frac{N}{b}$ but stayed in the range $[4542, 8582]$ and turned out to be a small fraction of the total number of leaves (see Table 2). We can therefore state that SB-trees are very useful in practice when the number of suffixes to be inserted does not exceed the number of SB-tree leaves. Otherwise, the whole reconstruction takes as much time as updating.

In our experiments, we also wanted to find out how fast the SB-tree insertion procedure was. We therefore started out by comparing SB-trees to UNIX Prefix B-Trees [4] and took about 240 text megabytes from the AT&T phone book. We chose $m \approx 3.9$ million suffixes and inserted the 10-grams associated with the suffixes in SUF_m (i.e., their only first 10 characters) into an empty SB-tree. We then performed the same insertion into an empty Prefix B-Tree. We found out that inserting into the SB-tree is *five times* faster than inserting into the Prefix B-tree (approximately 30 minutes versus 2.5 hours). Moreover, our SB-tree occupied slightly less space than a Prefix-B-tree even in our 12.3-per-indexed-suffix implementation. The problem posed by Prefix B-trees regards information duplication, whereas SB-trees exploited lexicographic order better. We expect to save much more space for larger values of q and N.

Finally, we compared SB-trees to the "start-over" solution obtained by merging two suffix arrays which stored SUF_m and all the suffixes kept in the SB-tree leaves, respectively. We set the

[4] We used the efficient implementation by P. Weinberger at AT&T [32].

page size to $B = 1$ kilobytes and took $m = 10^4$ with $N = 32$ and 64 million suffixes. In this case, inserting into the SB-tree usually went twice as fast as merging the two suffix arrays. We expect to save even more time for larger N's; however, if m approaches $\frac{N}{b}$, merging suffix arrays takes as much time as updating SB-trees (as previously verified for SB-tree reconstruction).

5 Conclusions and Further Research

In this paper, we presented an efficient implementation of the SB-tree data structure. A large set of experiments confirmed our theoretical predictions: SB-trees lead to very fast searches and can be updated in a reasonable amount of time. In the light of their good theoretical bounds and very good practical performance, we believe that SB-trees can be a very significant tool in practical applications.

As far as further research goes, we suggest using the I/O interface provided by the Transparent Parallel I/O Environment (TPIE) [30] to study the buffering strategy's influence on SB-trees' overall performance. We believe that particular attention should be given to obtaining a good tuning between the page size and the string length in order to improve insertion performance (see Section 4.2). We also believe that the following topics deserve further study and experimentation:

- *Static Text.* Space saving is also important for a *static* text. For example, let us take a text collection stored on a CD-ROM and assume that we want to keep its index on some other CD-ROMs. The main problem is how to limit their number. This, in turn, involves the problem of reducing the index space as much as possible while maintaining acceptable performance. If the set is fixed, we can simplify the SB-tree structure a great deal as shown in Section 3.1. We can also store the SB-tree in a heap-like fashion, and therefore make the retrieval of an internal node's child possible by simple arithmetic operations. We suggest studying the time/space trade-off achievable by this simplified SB-tree in order to evaluate the influence of parameter k on the search and update performance (see Section 3.1).
- *Databases.* Text indexing data structures have a new application to databases with variable-length records, since they *efficiently* maintain the lexicographic order among byte sequences (the record file) under record insertions and deletions. It is possible to maintain several indices on the same database *without* duplicating its (multiple) keys. This is important in *compound attribute* organizations [19, Sect 6.5] for maintaining the lexicographic order of combined attributes of records without having to make (multiple) copies of them. Our idea is to consider each variable-length record in the database as a text string. With respect to Prefix B-trees that introduce key duplication, SB-trees use lexicographic order better and take advantage of the longest common prefix of two consecutive keys. Our experiments showed that SB-tree updating is five times faster than UNIX Prefix B-tree updating while the SB-tree takes up even less space. Commercial databases could therefore take advantage of this new technology.
- *Amortized insertion.* It does not seem likely that the $O(m \log_B(N+m))$ worst-case insertion complexity can be improved because it matches the worst-case bound achieved for inserting m *standard* keys in a regular B-tree (in the worst case, each key must be stored in a distinct leaf). However, the experimental results showed that the optimal worst-case complexity is too pessimistic and hence, it would be very useful to devise a faster *amortized* solution.

Acknowledgments.

We are particularly indebted to Ken Church, who invited us to visit the AT&T Bell Labs and provided us with the text archives and code for UNIX B-trees [32]. Without his support and precious suggestions, the experiments described in this paper would not have been possible. We

also thank Jeffrey Vitter, who showed us how to transform the amortized analysis for the update operations in [13] into a worst-case one. We also warmly thank Brenda Baker, Dan Caldwell, Raffaele Giancarlo, Mike Goodrich, Rao Kosaraju, Fabrizio Luccio, Doug McIlroy, Christos Polyzois, and Manolis Tsangaris for their numerous helpful comments on the early results of this paper.

References

[1] Amir, A., Farach, M., Galil, Z., Giancarlo, R., and Park, K. Dynamic dictionary matching. *Journal of Computer and System Science 49* (1994), 208–222.

[2] Andersson, A., and Nilsson, S. Efficient implementation of suffix trees. *Software–Practice and Experience 25*, 2 (1995), 129–141.

[3] Bairoch, A. PROSITE: a dictionary of sites and patterns in proteins. *Nucleic Acids Research 20* (1992), 2013–2018.

[4] Baker, B. S. A theory of parameterized pattern matching: Algorithms and applications. In *ACM Symposium on Theory of Computing* (1993), pp. 71–80.

[5] Bayer, R., and McCreight, C. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (1972), 173–189.

[6] Bayer, R., and Unterauer, K. Prefix B-trees. *ACM Trans. Database Syst. 2*, 1 (1977), 11–26.

[7] Church, K. W., and Rau, L. F. Commercial applications of natural language processing. *Communications of the ACM 38* (1995), 71–79.

[8] Clark, D. R., and Munro, J. I. Efficient suffix trees on secondary storage. In *ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 383–391.

[9] Cleary, J. G., and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications 32* (1984), 396–402.

[10] Comer, D. The ubiquitous B-Tree. *Computing Surveys 11* (1979), 121–137.

[11] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[12] Darragh, J. J., Cleary, J. G., and Witten, I. H. Bonsai: A compact representation of trees. *Software - Practice and Experience 23* (1993), 277–291.

[13] Ferragina, P., and Grossi, R. A fully-dynamic data structure for external substring search. In *ACM Symposium on Theory of Computing* (1995), pp. 693–702. Also: An external-memory indexing data structure with applications, TR 18/96, Dipartimento di Sistemi e Informatica, Università di Firenze (1996), submitted to journal.

[14] Fox, A. E. *et al. Communications of the ACM: Special issue on "digital libraries"* (1995), vol. 38.

[15] Frenkel, K. A. The human genome project and informatics. *Communications of the ACM 34* (1991), 41–51.

[16] Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992, ch. 5:"New indices for text: PAT trees and PAT arrays", pp. 66–82.

[17] Gusfield, D., Landau, G. M., and Schieber, B. An efficient algorithm for all pairs suffix-prefix problem. *Information Processing Letters 41* (1992), 181–185.

[18] Kephart, J., Sorkin, G., Arnold, W., Chess, D., Tesauro, G., and White, S. Biologically inspired defenses against computer viruses. In *International Joint Conference on Artificial Intelligence* (1995), pp. 1–12.

[19] Knuth, D. E. *The Art of Computer Programming.* Addison-Wesley, 1973, ch. 3: Sorting and Searching.

[20] Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22*, 5 (1993), 935–948.

[21] Maxam, A. M., and Gilbert, W. A new method for sequencing DNA. In *Proc. Natl. Acad. Sci. USA* (1977), pp. 560–564.

[22] McCreight, E. M. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2 (1976), 262–272.

[23] Mewes, H. W., and Heumann, K. Genome analysis: Pattern search in biological macromolecules. In *Combinatorial Pattern Matching* (1995), pp. 261–285.

[24] Moffat, A. Implementing the PPM data compression scheme. *IEEE Transactions on Communications 38* (1990), 1917–1921.

[25] Morrison, D. R. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15* (1968), 514–534.

[26] *The Oxford English Dictionary*, second edition, 1989.

[27] Patt, N. P. *IEEE Computer: Special Issue "The I/O Subsystem: A candidate for improvement"* (1994), vol. 27.

[28] Prywes, N. S., and Gray, H. J. The organization of a Multilist-type associative memory. *IEEE Trans. on Communication and Electronics 68* (1963), 488–492.

[29] Shang, H. *Trie methods for text and spatial data structures on secondary storage.* PhD thesis, McGill University, 1995.

[30] Vengroff, D. E. A transparent parallel I/O environment. In *DAGS Symp. on Parallel Comp.* (1994).

[31] Vitter, J. S. *Algorithmica: Special Issue on "Large-Scale Memories"* (1994), vol. 12.

[32] Weinberger, P. J. Unix b-trees. AT&T Bell Laboratories (personal communication).

[33] Weiner, P. Linear pattern matching algorithm. In *IEEE Symp. on Switching and Automata Theory* (1973), pp. 1–11.

[34] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Info. Theory 23* (1977), 337–343.

[35] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theory 24* (1978), 530–536.