A Data Structure for a Sequence of String Accesses in External Memory

VALENTINA CIRIANI

University of Milano

PAOLO FERRAGINA AND FABRIZIO LUCCIO

University of Pisa

AND

S. MUTHUKRISHNAN

Rutgers University

Abstract. We introduce a new paradigm for querying strings in external memory, suited to the execution of sequences of operations. Formally, given a dictionary of n strings $S_1, \ldots, S_n$, we aim at supporting a search sequence for m not necessarily distinct strings $T_1, T_2, \ldots, T_m$, as well as inserting and deleting individual strings. The dictionary is stored on disk, where each access to a disk page fetches B items, the cost of an operation is the number of pages accessed (I/Os), and efficiency must be attained on entire sequences of string operations rather than on individual ones.

Our approach relies on a novel and conceptually simple self-adjusting data structure (SASL) based on skip lists, that is also interesting *per se*. The search for the whole sequence $T_1, T_2, \ldots, T_m$ can be done in an expected number of I/Os:

$$O\left(\sum_{j=1}^{m} \frac{|T_j|}{B} + \sum_{i=1}^{n} \left(n_i \log_B \frac{m}{n_i}\right)\right),$$

where each T_j may or may not be present in the dictionary, and n_i is the number of times S_i is queried (i.e., the number of T_js equal to S_i). Moreover, inserting or deleting a string S_i takes an

This work was partially supported by the Italian MIUR project ALGO-NEXT, and by the MIUR FIRB Project Italy-Israel on "Pattern matching and discovery algorithms on discrete structures, with applications to Bioinformatics."

Part of this work appeared in a preliminary version in the *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002, 219–227.

Authors' addresses: V. Ciriani, Department of Information Technology, University of Milano, Via Bramante 65, 126013 Crema; P. Ferragina, F. Luccio (contact author), Department of Computer Science, Universita di Pisa, Largo Pontecorro 3, 156127, Pisa, e-mail: luccio@di.unipi.it; S. Muthukrishnan, Department of Computer Science, Rutgers University, 110 Frelinghuysen Rd., Piscataway, NJ 08854.

expected amortized number $O(\frac{|S_i|}{B} + \log_B n)$ of I/Os. The term $\sum_{j=1}^{m} \frac{|T_j|}{B}$ in the search formula is a lower bound for reading the input, and the term $\sum_{i=1}^{n} n_i \log_B \frac{m}{n_i}$ (entropy of the query sequence) is a standard information-theoretic lower bound. We regard this result as the *static optimality theorem for external-memory string access*, as compared to Sleator and Tarjan's classical theorem for numerical dictionaries [Sleator and Tarjan 1985]. Finally, we reformulate the search bound if a cache is available, taking advantage of common prefixes among the strings examined in the search.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching; sorting and searching*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Design

Additional Key Words and Phrases: Skip list, external-memory data structure, sequence of string searches and updates, caching

ACM Reference Format:

Ciriani, V., Ferragina, P., Luccio, F., and Muthukrishnan, S. 2007. A data structure for a sequence of string accesses in external memory. ACM Trans. Algor. 3, 1, Article 6 (Feb. 2007), 23 pages. DOI = 10.1145/1186810.1186816 http://doi.acm.org/10.1145/1186810.1186816.

1. *Introduction*

While conventional datastores primarily manage numerical data or fixed-length keys, new warehouses are increasingly storing massive alphanumeric strings of varying length. In addition to string data proliferation, more and more transactions for searching, adding, deleting, or modifying strings must be handled. As a result, the problem of managing massive string data under large numbers of transactions is emerging as a fundamental challenge. Many different sources of string data exist. Patent databases contain full texts of patent documents, online libraries, biological databases, product catalogs, web-server logs, and other Internet traffic data, customer feedback and annotations, etc. A typical and intriguing example arises with XML files, the emerging standard for publication and interchange of heterogeneous, incomplete, and/or irregular data over the Internet. XML systems archive vast amounts of texts and face voluminous queries whose evaluation involves navigating through a tree, or possibly a graph. The current approach consists of encoding document paths into strings of arbitrary length (e.g., `book/author/name/`) and replacing tree navigational operations with string prefix queries [Aboulnaga et al. 2001; Cooper et al. 2001]. This requires managing massive textual data and processing a stream of string queries of arbitrary lengths in external memory [Ailamaki et al. 1999].

String algorithms and data structures generally support operations *individually*, and are designed for attaining high efficiency in the worst case. Major examples are related to the use of suffix trees and suffix arrays [Gusfield 1997]. However, there is a need for dictionary solutions that are efficient on an entire sequence of string transactions and possibly adapt themselves to a time-varying distribution of string queries, rather than being effective on the worst-case complexity of a single operation. For the first time in the literature (see Kärkkäinen and Rao [2003] and Ferragina [2005]), we seek solutions that are competitive or optimal over the entire sequence of operations in the framework of amortized analysis, underscoring the fact that the worst-case cost over a sequence of operations may be far less than the sum of the worst-case costs of single operations in the sequence.

The idea of supporting sequences of queries is, in fact, not new. A classic paper [Sleator and Tarjan 1985] presented an adaptive data structure for numeric-valued dictionaries that automatically reorganizes itself in response to a sequence of accesses. They introduced the *splay* operation for binary search trees (combinations of node rotations that move the accessed node to the root) and proved the *Static Optimality theorem*, which states that the total time required by m accesses to an n-node splay tree is given by:

$$O\left(m + \sum_{i=1}^{n} n_i \log(m/n_i)\right),$$

where n_i is the number of occurrences of the ith item in the query sequence. If computed as the sum of the worst-case costs per operation, the total time would be $\Theta(m \log n)$, while a better bound is obtained if the sequence of accesses is skewed. The splay tree is in fact as efficient as any fixed search tree, since the aforementioned access cost matches the information-theoretic lower bound for the total access time of any *fixed* search tree, including one that knows the sequence of accesses ahead of time [Abramson 1983; Sleator and Tarjan 1985]. The term $\sum_{i=1}^{n} n_i \log(m/n_i)$ relates to the entropy of the sequence of accesses. An amortized solution for internal, memory string access using lexicographic trees was presented in the same work, but the authors left open the problem of designing a self-adjusting, B-degree search structure for strings, which has thus far remained open. This is what we treat here.

A computer is abstracted as consisting of an internal memory and a disk. Accesses to the latter fetch B-sized pages, and the cost of a sequence of operations is given by the number of disk accesses, hereafter called I/Os [Vitter 2002]. Arbitrarily long strings may not fit in single disk pages, and each string comparison might need many disk accesses if executed in a brute-force manner. On these grounds, based on a study of data structures for massive transaction on string dictionaries, we prove the first-known static optimality theorem in this model. Specifically, given a dictionary $\mathcal{D}$ of n strings $S_1, \ldots, S_n$ of total length N, we propose a self-adjusting data structure for $\mathcal{D}$ in which a search sequence for m (not necessarily distinct) strings $T_1, T_2, \ldots, T_m$ can be completed in

$$O\left(\sum_{j=1}^{m} \frac{|T_j|}{B} + \sum_{i=1}^{n} \left(n_i \log_B \frac{m}{n_i}\right)\right)$$

expected number of I/Os, where each T_j may or may not be present in the dictionary, and n_i is the number of times S_i is queried (i.e., the number of T_js equal to S_i). The data structure occupies a total of $O(N/B)$ expected disk pages. We also show that inserting or deleting a string S_i takes $O(\frac{|S_i|}{B} + \log_B n)$ expected amortized number of I/Os.

Clearly, $\sum_{j=1}^{m} \frac{|T_j|}{B}$ disk accesses are needed to read the query sequence, while the term $\sum_{i=1}^{n} n_i \log_B \frac{m}{n_i}$ is related to the entropy of the query sequence, and a standard information-theoretic lower bound [Mehlhorn 1984]. The B-way branching provided by the disk page size is fully utilized in that the total size of the query strings is divided by B, and the logarithm in the entropy term is taken to the base B. In other words, our data structure is as efficient as any fixed search tree, since the access cost matches the information-theoretic lower bound—in terms of number of disk-page accesses—of *any fixed* search tree, including one that knows the sequence of accesses ahead of time [Abramson 1983; Sleator and Tarjan 1985]. The total bound is analogous to that in Sleator and Tarjan's Static Optimality theorem [1985], but for the first time has been generalized to *external-memory string access*. Note also that the space bound is asymptotically optimal.

A natural approach for attaining comparative results could be designing a splaying operation for a suitable external-memory string data structure. For this purpose, however, the only known candidate is the String B-tree [Ferragina and Grossi 1999], whose structural constraints keep it balanced in the presence of inserts and deletes. These constraints should be maintained under the sophisticated splaying operations known for multiway search trees [Sherk 1995], resulting in a complicated structure that ultimately would not provide a result comparable to the preceding optimality theorem. In fact, String B-trees exhibit the best worst-case known performance per string operation, but their use on a query sequence gives an I/O-bound of $O(\sum_{j=1}^{m}(\frac{|T_j|}{B}) + m \log_B n)$. This I/O-bound is not optimal over the whole sequence, and might be significantly higher than the aforementioned bound on repetitive query sequences.

To get our result, we have designed a new data structure, called *self-adjusting skip lists* (SASL), starting from the well-known skip lists proposed for dictionaries of numerical values in internal memory [Pugh 1990]. Skip lists maintain items in levels, each level consisting of a randomly chosen half of items from the previous level. We extend this mechanism to work in external memory by choosing items with probability $\Theta(1/B)$ at each level to populate the next level. We also independently extend it to work with strings, rather than numbers. The main challenge is to render the data structure self-adjusting under these two simultaneous extensions. In particular, when a string is accessed, certain items must be *promoted* up the levels and simultaneously other items must be *demoted* down the levels, guaranteeing locality of reference in the memory accesses. A technical novelty with SASL is a simple randomized demotion strategy that, together with the doubly-exponential grouping of the skip list levels, guides demotion and guarantees locality of reference such that frequent items remain at the highest levels and effective I/O-bounds are achieved. A caveat is that our results are attained combining the basic properties of skip lists with the randomized demotion strategy of SASL, and hold with high probability. This is a departure from Sleator and Tarjan's [1985] approach, where the bounds are deterministic.

SASLs are very simple and exhibit many properties that standard skip lists and other self-adjusting data structures also possess. They achieve balance without explicit weight constraints and are never worse than String B-trees on search operations, but can be significantly better if the sequence of queries is highly skewed and changes over time, as most transactions do in practice. On individual operations, SASLs require $O(\log_B N)$ I/Os with high probability, whereas String B-trees guarantee worst-case bounds. In addition, a very simple SASL for self-adjusting internal-memory string access is directly obtained by setting $B = \Theta(1)$. This is a simplification over the best-known self-adjusting in-memory data structure for strings [Sleator and Tarjan 1985].

The rest of this article is organized as follows. In Section 2 we review standard skip lists and other related work. In Section 3 we introduce SASL and its basic properties, and the search and update algorithms for this structure. Also in this section, SASL is studied in internal memory, as we believe this is of independent interest. In Section 4 we extend SASL to external memory and derive the Static Optimality theorem for external-memory string dictionaries. In Section 5 we combine the use of SASL with a caching strategy and reformulate the search bound, exploiting the existence of common prefixes among the strings examined in the search. Since our article is the first to study self-adjusting data structures in external-memory string dictionaries, several problems remain to be explored, as discussed in Section 6.

2. *Skip Lists and Related Work*

Skip lists, the data structure from which we start, were introduced by Pugh [1990] as a simple alternative to balanced search trees for managing dictionaries of atomic items for which a total ordering relation $\leq$ is defined. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be a totally ordered set of n items. For each item s_j, we flip a (fair) coin until a tail comes up. According to the coin-flipping process, we construct a sequence of sets $\mathcal{S}_1, \ldots, \mathcal{S}_h$ such that $\mathcal{S}_h = \mathcal{S}$, and $\mathcal{S}_{h-i+1}$ is the set of all items for which we flipped the coin at least i times. Notice that $\mathcal{S}_j \subseteq \mathcal{S}_{j+1}$, unlike in the original article [Pugh 1990], where lists were numbered from largest to smallest. The *height* h is a random variable which depends on the outcomes of the coin flips; it can be easily proved that $\mathbb{E}[h] = O(\log n)$ [Pugh 1990]. The skip list $\mathcal{L}$ for $\mathcal{S}$ consists of a collection of lists, organized as follows.

—Each list $\mathcal{L}_i$ is doubly-linked and contains the elements of $\mathcal{S}_i \cup \{-\infty, +\infty\}$ stored in increasing order; and

—each item in $\mathcal{L}_i$ has a "vertical" pointer, pointing "down" to its occurrence in $\mathcal{L}_{i+1}$.

The expected size of each list grows exponentially with i, in fact, $\mathbb{E}[|\mathcal{L}_i|] = \Theta(2^i)$. The basic operation is *searching for an item x in $\mathcal{S}$*, which is performed by traversing the skip list per increasing level, starting from $\mathcal{L}_1$ at $-\infty$. Each list $\mathcal{L}_i$ is scanned rightwards until two contiguous items y_j, y_{j+1} are encountered such that $y_j < x \leq y_{j+1}$. If $y_{j+1} = x$, then the search is successful and $\mathcal{L}_i$ is the first list containing item x. Otherwise, the search moves to the occurrence of y_j in the next list $\mathcal{L}_{i+1}$ and the rightwards traversal is repeated. When $i > h$, the search is deemed unsuccessful. Insert and delete operations are done likewise. Search, insert, and delete take $O(\log n)$ expected time [Pugh 1990].

Skip lists are not self-adjusting, therefore, a frequently accessed item might belong to the lowest list $\mathcal{L}_h$ and thus have expected searching time $O(\log n)$, with no differences from rarely looked-up items of $\mathcal{S}$. A self-adjusting skip list should "artificially" promote frequently accessed items to the highest levels of $\mathcal{L}$, facilitating their retrieval. This has been achieved in various ways [Ergun et al. 2001; Mulmuley 1994; Mehlhorn and Naher 1992], either assuming that the frequency of accesses are known or under specific assumptions, such as the probability of accesses being nonincreasing over time. In particular, the *biased* skip list (BSL) was introduced in Ergun et al. [2001] as a self-adjusting form of the classical skip list conceived for internal memory use. Here, each item has an associated an *MTF-rank* (move-to-front rank) which determines its level in the underlying skip list. The smaller the MTF-rank, the higher the level to which the item belongs. BSL supports search/insert/delete operations in $O(\log_2 r)$ expected amortized time, where r is the

MTF-rank of the queried item (we are suppressing certain details not pertinent to us here). A query on an item having rank r promotes the accessed item to the top level of the skip list, sets its MTF-rank to 1, demotes $\Theta(\log_2 r)$ items to lower levels (selected according to current MTF-ranks), and increments the $\Theta(r)$ MTF-ranks of all items whose rank was smaller than r. The recomputation of these $\Theta(r)$ MTF-ranks is done implicitly by means of an additional MTF-list.

Since we are interested in an external-memory variant of the self-adjusting skip list, we must try to get locality of reference in BSL. All natural adaptations of known strategies, however, seem to fail. Consider, for example, applying the bucketing strategy of Callahan et al. [1995] on BSL, which would now consist of $O(\log_B n)$ levels. After a query operation, both the BSL and MTF-list must be consistently updated as a result of the promotion/demotion steps. This is hard to do in an I/O-efficient manner, since the BSL and MTF-list are ordered according to two different criteria (the former by item value, the latter by MTF-rank). In particular, if we keep *crosspointers* between an element of the MTF-list and its copy in the BSL, and vice versa, we have immediate access to the $\Theta(\log_B r)$ items to be demoted, but their movement to lower levels in the skip list induces split/merge operations on the disk pages allocated to these lists. A split/merge of one page needs the modification of $\Theta(B)$ crosspointers, namely, those pointing to items of the BSL belonging to the affected page and coming from arbitrary disk pages of the MTF-list. Hence, each split/merge operation on the BSL may require $\Theta(B)$ I/Os, thus resulting in a $\Theta(B \log_B r)$ I/O-bound for the search operation (which is, of course, suboptimal). All other natural strategies seem also to lead to suboptimal bounds and complicated BSL management. Moreover, note that the previous considerations apply to dictionaries of numeric (or atomic) values, while string dictionaries are even harder to handle, as their string entries may be arbitrarily long.

In what follows, we introduce a variation of skip lists for the external-memory string dictionary problem, called self-adjusting skip lists (SASL). SASL is a cascade of skip lists of increasing sizes, growing doubly exponentially. As a result of a search/update operation, SASL is adjusted with proper *deterministic promotion* and *randomized demotion* strategies, thus achieving locality of reference, supporting the search for an item in $O(\log_B r)$ expected I/Os, and updating the structure in $O(\log_B n)$ expected amortized I/Os.

3. *Self-Adjusting Skip Lists (SASL) in Internal Memory*

As anticipated, an SASL $\mathcal{L}$ is a skip list with a self-adjusting property. $\mathcal{L}$ will contain random subsets of items in successive levels for balance, and will force frequently accessed items to reside in higher levels to facilitate their future retrieval. In addition, all operations on $\mathcal{L}$ will have to perform efficiently in external memory under blocked accesses.

For clarity, we start studying SASL in internal memory in this section, keeping in account all the preceding goals. The first key idea in SASL is to structure the *column* of each item x, that is, the list of all occurrences of x linked via the "vertical" pointers, by means of two parts: one *randomized* and one *deterministic*. The former is generated as in the original skip list via coin tosses, while the deterministic part adapts itself continuously to the sequence of searches/updates. The second key idea in SASL is to force this behavior to follow a *double exponential cascade* of skip lists,[1] driven by proper deterministic promotion and randomized demotion strategies for guaranteeing locality of reference. We will show in the next subsections that determinism will not jeopardize the nice properties "in expectation" (and "with high probability") of the original skip lists. It will additionally provide us with better control of the features discussed previously.

3.1. THE STRUCTURE OF SASL. Given a set $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$, a SASL $\mathcal{L}$ for $\mathcal{S}$ is formed by $H = \Theta(\log n)$ lists $\mathcal{L}_1, \dots, \mathcal{L}_H$, satisfying the property $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$, with $\mathcal{L}_i$ containing $\Theta(2^i)$ items of $\mathcal{S}$ for $1 \leq i \leq H - 1$, and $\mathcal{L}_H$ containing all the items of $\mathcal{S}$. The novelty of SASL is that the items are distributed among the lists via a special strategy which combines deterministic and randomized coin tosses, and is further driven by the following organization.

The lists are logically grouped into horizontal *bands* $\mathcal{B}_1, \mathcal{B}_2, \dots$ whose sizes increase exponentially, corresponding to the cascade of skip lists mentioned earlier. Each band $\mathcal{B}_i$ consists of the 2^{i-1} adjacent lists $\mathcal{L}_{2^{i-1}} \cdots \mathcal{L}_{2^i-1}$. So, $\mathcal{B}_1$ consists of the only list $\mathcal{L}_1$, $\mathcal{B}_2$ consists of $\mathcal{L}_2$ and $\mathcal{L}_3$, etc. The number of bands is $b(n) = \Theta(\log \log n)$. This logical decomposition is used to drive the distribution of items of $\mathcal{S}$ among the various lists, always guaranteeing the condition that $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$. Recall that we number the lists from top to bottom, and then we say that a list $\mathcal{L}_i$ is *higher* than $\mathcal{L}_{i+k}$ (or $\mathcal{L}_{i+k}$ is *lower* than $\mathcal{L}_i$) for any positive k. We define the *height* H_i of a band $\mathcal{B}_i$ as the total number of lists belonging to the lower bands $\mathcal{B}_{i+1}, \dots, \mathcal{B}_{b(n)}$, that is, $H_i = 2^{b(n)} - 2^i$. The height H_1 of the highest band $\mathcal{B}_1$, plus 1 for its own list, gives the total height $H = \Theta(\log n)$ of $\mathcal{L}$.

We associate two integer quantities with each item $s \in \mathcal{S}$; namely, a random value $r(s)$ indicating the number of times heads came out in a sequence of coin tosses on s, and a deterministic value $d(s)$ equal to the height H_i of some band $\mathcal{B}_i$. In this scheme, $d(s)$ represents a deterministic promotion of s to the band $\mathcal{B}_i$, from whence it may proceed to higher lists via a random promotion $r(s)$. The height of item s is thus defined as $h(s) = r(s) + d(s)$. We bound $h(s)$ to be (at most) H, by setting $h(s) = H$ if $r(s) + d(s) > H$. Item s is allocated in the list $\mathcal{L}_{H-h(s)+1}$, and in all the lists below it up to $\mathcal{L}_H$. Observe that the deterministic part of a column cannot end inside a band, but always synchronizes to its beginning. As we shall see, the value of $d(s)$ changes over time as the search/update operations are executed so as to adapt $\mathcal{L}$ to the observed frequencies of the accessed items. The truncation of $h(s)$ to H will not affect our mathematical analysis because it can be proved by standard techniques [Pugh 1990] that the number of truncated columns is a constant, so they can be managed specifically. A simple example of SASL built for a set $\mathcal{S}$ of numerical items and for internal memory is shown in Figure 1.

Based on the preceding, we introduce the following terminology:

(1) An item s *resides* in a band $\mathcal{B}_i$ if $d(s) = H_i$. That is, the deterministic part of its column determines the residence of s. As already noted, the self-adjusting strategy may keep the value of $d(s)$ changing, hence the residence of s may change. In the example of Figure 1, item 8 resides in $\mathcal{B}_1$ and item 5 resides

[1]A loosely similar structuring into a cascade of search data structures (e.g., AVL trees) of doubly-exponentially increasing size was used in Iacono [2001]. However, in that paper, the author addressed different problems in internal memory, and adopted a deterministic MTF-strategy to promote and demote items between data structures. The comments on BSL raised in the preceding section apply to this approach, too.

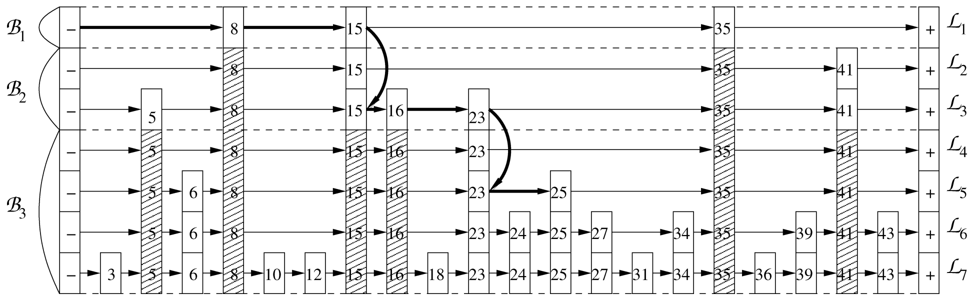

FIG. 1. An internal-memory SASL for numerical items, decomposed into three bands $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ containing one, two, and four lists respectively. The deterministic part of each column is shadowed. Bold arrows denote the links traversed during the search for item 25.

in $\mathcal{B}_2$. Note that the deterministic parts of the columns of items residing in $\mathcal{B}_i$ pass through $\mathcal{B}_{i+1}, \ldots, \mathcal{B}_{b(n)}$. Parameter β_i shall denote the number of items residing in $\mathcal{B}_i$.

(2) An item s *appears* in the band $\mathcal{B}_i$ if it resides in a lower band $\mathcal{B}_j$, with $j > i$, and its height $h(s)$ is greater than H_i (thus, the randomized part of its column passes through band $\mathcal{B}_i$). In the example of Figure 1, item 15 resides in $\mathcal{B}_2$ and appears in $\mathcal{B}_1$. A random variable α_i shall denote the number of items which appear in $\mathcal{B}_i$.

At a first glimpse, it might seem that a deterministic promotion strategy would jeopardize the nature itself of skip lists. However, by further constraining the number of items residing in a band, we can actually prove that every band behaves as if it were a *single skip list*, thus inheriting all the nice properties of this data structure. In addition, the deterministic part of every item's column acts as an anchor between adjacent bands, a useful feature for making incremental searches among them (through fractional cascading [Chazelle and Guibas 1986]). More precisely, the number of items residing in a band $\mathcal{B}_i$ is forced to be $\beta_i = 2^{2^{i-1}}$, that is, it grows doubly exponentially in band number. In the lowest band, we have $\beta_{b(n)} = O(2^{2^{b(n)-1}})$, with $\beta_{b(n)}$ at most equal to $2^{2^{b(n)-1}}$.

For the sake of exposition, we were silent on the formal definition of $b(n)$. We now define $b(n)$ exactly as the minimum number of bands of doubly-exponential size where n objects are residing. Clearly, $b(n) = \Theta(\log\log n)$. In the example of Figure 1, we have $\beta_1 = 2^{2^{1-1}} = 2$ items residing in $\mathcal{B}_1$, $\beta_2 = 2^{2^{2-1}} = 4$ items residing in $\mathcal{B}_2$, and $\beta_3 = 14 < 2^{2^{3-1}} = 16$ items residing in $\mathcal{B}_3$.

Of course, an item s can appear in more than one band if $r(s)$ is sufficiently large. In fact, the column of an item may pass through a given band $\mathcal{B}_i$ because it has a long deterministic part (captured by β_i) or has a long randomized part (captured by α_i, see point 2, given earlier). Since the number of lists in a band $\mathcal{B}_j$ is an exponential function of j, we can expect the random variable α_i, with $i < j$, to have a low (in fact, constant) expected value. In other words, the bands below $\mathcal{B}_i$ do not contribute significantly to the number of items appearing in $\mathcal{B}_i$. Formally we have:

THEOREM 3.1. *The expected number of items residing or appearing in the band $\mathcal{B}_i$, with $i < b(n)$, is $\mathbb{E}[\alpha_i + \beta_i] \leq 2 + \beta_i$.*

PROOF. Recall that the lists and bands are numbered from top (smallest) to bottom (largest). Let the random variable $\alpha_{j,j+k}$ represent the number of items residing in $\mathcal{B}_{j+k}$, with $k > 0$, and appearing in the higher band $\mathcal{B}_j$. Each of them, say, item s, has a random part $r(s)$ spanning the bands $\mathcal{B}_{j+k}, \mathcal{B}_{j+k-1}, \dots, \mathcal{B}_{j+1}$, hence we have $r(s) > H_j - H_{j+k} = (2^{b(n)+1} - 2^j) - (2^{b(n)+1} - 2^{j+k}) = 2^j(2^k - 1)$. The probability of this event $\mathbb{P}(r(s) > H_j - H_{j+k})$ is therefore less than $(1/2)^{2^j(2^k-1)}$. Summing up on all bands which lie below $\mathcal{B}_j$, and recalling that the number of items residing in $\mathcal{B}_{j+k}$ is $\beta_{j+k} \le 2^{2^{j+k-1}}$, we get: $\mathbb{E}[\alpha_j] = \mathbb{E}[\sum_{k=1}^{b(n)-j} \alpha_{j,j+k}] = \sum_{k=1}^{b(n)-j} \mathbb{E}[\alpha_{j,j+k}] \le \sum_{k=1}^{b(n)-j} \beta_{j+k} \times (1/2)^{2^j(2^k-1)} \le \sum_{k=1}^{b(n)-j} (1/2)^{2^j(2^{k-1}-1)} \le 2$. $\square$

We will show in Section 4 that the deterministic columns passing through a band play no role in the search operation. Then, from Theorem 3.1, we have that each band can be seen as an independent skip list with all the nice structural properties of such lists. Furthermore, since there are $\sum_{k=1}^{i-1} \beta_k = \Theta(\beta_i)$ deterministic columns passing through $\mathcal{B}_i$, the extra space occupied by all the deterministic columns is asymptotically no more than the space occupied by classical skip lists and can actually be bounded as $O(\sum_{i=1}^{b(n)} \beta_i \times H_i \times \log n) = O(n \log n)$ bits, as in the original skip lists.

The description of SASL given thus far has been simplified for clarity, let us call it a *simple* SASL. However, the actual structure may be more complicated if $\beta_{b(n)}$, that is, the number of items residing in the lowest band $\mathcal{B}_{b(n)}$, is small. In particular, if $\beta_{b(n)} = o(2^{2^{b(n)-1}})$, the number of items allocated in $\mathcal{L}$ is $\sum_{i=1}^{b(n)-1} 2^{2^{i-1}} + o(2^{2^{b(n)-1}})$, which is much smaller than the total number $2^{b(n)-1} \sum_{i=1}^{b(n)-1} 2^{2^{i-1}}$ of deterministic column elements present in $\mathcal{B}_{b(n)}$, thereby making the memory requirement and search time unnecessarily big. Therefore, the general structure of a SASL $\mathcal{L}$ is as follows.

$\mathcal{L}$ consists of two parts: $\mathcal{L}_{up}$ and $\mathcal{L}_{down}$. If $\beta_{b(n)}$ is "large" enough (see the following), then $\mathcal{L}_{down}$ is empty and $\mathcal{L}_{up}$ is precisely the simple SASL described before. If $\beta_{b(n)}$ is "small," then $\mathcal{L}_{up}$ is a SASL containing the previous bands $\mathcal{B}_{b(1)} \dots \mathcal{B}_{b(n-1)}$, and $\mathcal{L}_{down}$ is a standard skip list containing items residing in the last band $\mathcal{B}_{b(n)}$, without the deterministic parts of the columns of items residing in upper bands. The concept of $\beta_{b(n)}$ as being large or small must be flexible, as the size of $\mathcal{L}$ changes continuously under insertions and deletions of items. We adopt the following strategy: If $\beta_{b(n)} > \frac{2}{3} 2^{2^{b(n)-1}}$, then $\mathcal{L}_{down}$ is empty and $\mathcal{L} = \mathcal{L}_{up}$. If $\beta_{b(n)} < \frac{1}{3} 2^{2^{b(n)-1}}$, then $\mathcal{L}_{up}$ is nonempty and $\mathcal{L}$ is divided into two portions, $\mathcal{L}_{up}$ and $\mathcal{L}_{down}$, as specified previously. Otherwise, $\frac{1}{3} 2^{2^{b(n)-1}} \le \beta_{b(n)} \le \frac{2}{3} 2^{2^{b(n)-1}}$, either one of the two solutions may be adopted, depending on the updates of $\mathcal{L}$. The SASL migrates from one configuration into the other as items are inserted and deleted from $\mathcal{L}$. The cost of this migration is amortized by the cost of the $\Theta(2^{2^{b(n)-1}})$ update operations executed to trigger either one of the two aforementioned special cases (see Section 3.3).

The initial construction of a SASL $\mathcal{L}$ for n items is performed as follows. Let k be the greatest integer such that $n = \sum_{i=1}^{k} 2^{2^{i-1}} + u$ with $0 \le u \le 2^{2^k}$. If $u \le 2^{2^k-1}$, then the two parts $\mathcal{L}_{up}$ and $\mathcal{L}_{down}$ of $\mathcal{L}$ are built, with $\mathcal{L}_{up}$ containing $\sum_{i=1}^{k} 2^{2^{i-1}}$ items randomly chosen from the initial n, and $\mathcal{L}_{down}$ containing the remaining u items. Otherwise, ($u > 2^{2^k-1}$) and only $\mathcal{L}_{up}$ is built, containing n items. While $\mathcal{L}_{down}$ is generated as a classical skip list, the construction of $\mathcal{L}_{up}$ requires fixing the number of items and lists in each band. In fact, each band $\mathcal{B}_i$ is built as a standard skip list with $2^{2^{i-1}}$ items residing therein, and is forced to contain 2^{i-1} lists. If an item residing in $\mathcal{B}_i$ rises for more than 2^{i-1} steps, it invades the next bands above (e.g., see item 15 in Figure 1). Any item rising above the top of $\mathcal{B}_1$ is stopped there. The construction is done incrementally, choosing items at random. First, we choose $2^{2^0} = 2$ items to be inserted in the first band $\mathcal{B}_1$, and build a skip list for them (i.e., we generate their random part). Then we chose $2^{2^1} = 4$ items to be inserted in $\mathcal{B}_2$, for which we build a skip list, where we also insert the deterministic parts (of height 2) of the two items in $\mathcal{B}_1$. Incrementally, we choose $2^{2^{i-1}}$ items to be inserted in $\mathcal{B}_i$, for which we build a skip list, where we also insert the deterministic parts (of height 2^{i-1}) of items residing in the upper bands. Finally, we build a skip list for the u items in $\mathcal{B}_n$, either as the last band of $\mathcal{L}_{up}$ (curbed at height $2^{b(n)-1}$, and with addition of the deterministic columns) or in $\mathcal{L}_{down}$.

3.2. THE SELF-ADJUSTING STRATEGY. While the search and update operations for SASL will be explained in the next section, we anticipate here the strategy to make the data structure self-adjusting. The classical search procedure on skip lists [Pugh 1990] is modified by moving items among the bands so that:

—The most frequently accessed items are promoted to higher bands for facilitating their future retrieval;
—the "population" of each band is kept almost unchanged along the sequence of queries; and
—the structural properties of SASL are preserved.

To this end, the access to an item x (residing in band $\mathcal{B}_j$) triggers the promotion of its residence to the highest band $\mathcal{B}_1$, and for each $i \in [1, j-1]$, forces demotion of the residence of one item x_i residing in band $\mathcal{B}_i$ to the immediately lower band $\mathcal{B}_{i+1}$. These operations are precisely defined as follows:

(1) An item x residing in $\mathcal{B}_j$ *is promoted* to $\mathcal{B}_1$ by increasing the value of $d(x)$ from H_j to H_1. In other words, the "deterministic part" of the column of x is extended through all bands up to $\mathcal{B}_2$, and the residence of x is raised to $\mathcal{B}_1$. The value $r(x)$ is preserved, to be used in case x is later demoted (see the next point, 2), but such a value is temporarily set to 1 to keep $r(x) + d(x) = H$.
(2) An item x residing in $\mathcal{B}_i$, $1 \leq i \leq b(n) - 1$, *is demoted* to $\mathcal{B}_{i+1}$ by decreasing the value of $d(x)$ from H_i to H_{i+1}. Specifically, the "deterministic part" of the column of x is shortened to one band down, the residence of x is moved down to $\mathcal{B}_{i+1}$, and the total height of its column is decreased by 2^i, that is, by the height of $\mathcal{B}_{i+1}$. The value of $r(x)$ remains unaltered unless a greater value was preserved for it at an earlier step. In this case, such a value is resumed, increasing the height of the column again. However, since such a height cannot exceed H, the value of $r(x)$ may be temporarily decreased again, preserving its actual value.

In the example of Figure 1, if item 25 is promoted to $\mathcal{B}_1$, the top of its column is lifted to the top of SALS, the current value $r(25) = 3$ is preserved for later use, and $r(25)$ is temporarily set to 1. If item 15 is demoted to $\mathcal{B}_3$, the top of its column is lowered by four positions. Note that $r(15) = 3$, however, since column 15 has its top in $\mathcal{B}_1$ before demotion, it may be the case that a higher value of $r(15)$ was preserved in a previous step and has to be resumed now, thus raising the top of the column again.

Changing the values $d()$ for a subset of $\Theta(j) = O(\log\log n)$ items implies updating the columns relative to these items. If the accessed item x comes from the list $\mathcal{L}_l$ in $\mathcal{B}_j$, we must insert x in all the $O(2^j)$ lists $\mathcal{L}_1, \ldots, \mathcal{L}_{l-1}$. Furthermore, we must remove each item x_i to be demoted from $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$, with $i \in [1, j-1]$, from exactly $H_i - H_{i+1} = 2^i$ lists. Once the items to be demoted have been found, their removal takes $\Theta(1)$ time per affected list due to the presence of "vertical" pointers between lists in adjacent levels of the SASL. Hence, the overall cost is $O(2^j + \sum_{i=1}^{j-1}(H_i - H_{i+1})) = O(2^j)$, that is, proportional to the width of the band $\mathcal{B}_j$ containing the accessed item.

A crucial problem is selecting uniformly at random the items to be demoted from the bands $\mathcal{B}_1, \ldots, \mathcal{B}_{j-1}$ by exploiting local information to work efficiently in external memory. This selection must also ensure that along the life of the SASL, the band $\mathcal{B}_j$ in which a searched item resides is not too deep in the cascade of skip lists, but its index j is related with the (unknown) probability of accessing the searched item. As attained in Ergun et al. [2001], the depth of the searched item is $\Theta(\log r)$, where r is its MTF-rank; however, we now achieve this result exploiting *locality of reference* in the memory accesses, still within high-probability bounds. Since skip lists inherently rely on randomization, additional randomization for making them self-adjusting does not affect the process significantly.

Our main idea is designing an algorithm which selects *uniformly at random* one of the items residing in any given band, without keeping extra pointers to identify the items residing in this band whose management might be costly in external memory (see Section 2). Hence, we introduce a set of local counters for driving random selection. For each item s of $\mathcal{L}$, we store a set of counters $c_i(s)$ at the top of the column of s, defined as follows:

Definition 3.2. Let the column of item s have its top in the band $\mathcal{B}_h$. For $k = 0, 1, \ldots, b(n) - h$, the *counter* $c_{h+k}(s)$ gives the total number of items (including s) that reside in the band $\mathcal{B}_{h+k}$ and are reachable from s via the classical skip list search procedure.

Note that if s resides in the band $\mathcal{B}_{h+r}$, $r \geq 0$, the counter $c_{h+r}(s)$ also counts s, while the items counted by all other $c_{h+k}(s)$ are located at the right of s, and their columns have height equal to or smaller than the column of s. Consider the example of Figure 1. Item 8 resides in $\mathcal{B}_1$. At the top of its column, we store $c_1(8) = 2$ (referring to items 8 and 35, but not to 15, which does not reside in $\mathcal{B}_1$), $c_2(8) = 3$ (items 15, 16, and 41), and $c_3(8) = 12$ (items 10 to 43 residing in $\mathcal{B}_3$). For item 15, we store $c_1(15) = 1$ (item 35), $c_2(15) = 3$ (items 15, 16, and 41), and $c_3(15) = 10$ (items 18 to 43 residing in $\mathcal{B}_3$). For item 5, whose top band is $\mathcal{B}_2$, we store $c_2(5) = 1$ (item 5 itself, but not items 15, 16, and 41, which are not reachable by a skip list search from 5), and $c_3(5) = 1$ (item 6). We also extend Definition 3.2 to $c_i(-\infty)$ and $c_i(+\infty)$, assuming that these two limiting items are residing in an ideal band $\mathcal{B}_0$, that is, their columns are totally deterministic (then, $c_i(-\infty) = \beta_i$ and $c_i(+\infty) = 0$). In the example, we have that $c_1(-\infty) = 2$, $c_2(-\infty) = 4$, $c_3(-\infty) = 14$.

To randomly select the item to be demoted from the band $\mathcal{B}_i$, we use the counters $c_i(s)$, whose values drive a random walk in $\mathcal{L}$ ending at an element on the lowest list of $\mathcal{B}_i$. The strategy is a direct extension of the search algorithm for skip lists and is implemented as follows:

Procedure RANDOM SELECT(i).
Input: An integer i, $1 \le i \le b(n)$.
Output: An item residing in $\mathcal{B}_i$ and chosen at random.

Method: Starting from the leftmost item $-\infty$ in the list $\mathcal{L}_1$, go right or down, flipping a weighted coin. Let s be the current item and s' be the item at its right (in the same list). If s' is not the top of its column, then go down, otherwise, go right with probability $c_i(s')/c_i(s)$ and go down with probability $1 - c_i(s')/c_i(s)$. Once the lowest list in $\mathcal{B}_i$ is reached, and as soon as going down is required, return the current item.

For example, apply RANDOM SELECT to the SASL of Figure 1 with input $i = 2$. The procedure will choose randomly between the items residing in $\mathcal{B}_2$: 5, 15, 16, 41. Starting with $s = -\infty$ in list $\mathcal{L}_1$, we have $s' = 8$, $c_2(-\infty) = 4$, and $c_2(8) = 3$. Hence, we may go right with probability 3/4 (towards items 15, 16, and 41) and down with probability 1/4 (towards item 5). Suppose the biased coin takes us to the right. We have $s = 8$, $s' = 15$, $c_2(8) = 3$, and $c_2(15) = 3$, hence we go right again with probability 1 (in fact, note that no item between 8 and 15 was residing in $\mathcal{B}_2$). We may then proceed, for example, with a down, right, and down move, to eventually select item 16. We have:

THEOREM 3.3. *The procedure* RANDOM SELECT *with input i returns an item chosen uniformly at random from among those residing in $\mathcal{B}_i$. The expected time is $O(2^i)$ and the extra space needed to store the auxiliary counters is $O(n \log n)$ bits, overall.*

PROOF. Due to the definition of $c_i(s)$, each move of RANDOM SELECT is done with a probability proportional to the number of possible target items that may be encountered. Hence the final item is hit uniformly at random. For proving the time bound $O(2^i)$, simply note that the path followed in the SASL for choosing an item x is the same path that would be followed in searching x in a skip list consisting of the top i bands. For proving the space bound, note that we store $O(\log \log n)$ counters $c_{h+k}(s)$ for each item s, each requiring $O(\log_2 c_{h+k}(s)) = O(\log_2 \beta_{h+k}) = O(2^{h+k})$ bits. We may use here, for example, a prefix-free variable-length encoding for integers. Summing up this space occupancy for $k = 0, \ldots, \log \log n$, due to the double-exponential nature of SASL, we have a total of $O(2^{b(n)}) = O(\log n)$ bits, which is sufficient to store all the counters relative to one item s. $\square$

From this proof, we note that the number of bits needed to store all the counters of an item is bounded above by the space needed to store a skip list pointer. So, our structuring does not introduce any overhead with respect to the classical skip list.

3.3. SEARCH AND UPDATE OPERATIONS. We have now all the tools for implementing the search and update operations for SASL in internal memory. The search for an item x starts in band $\mathcal{B}_1$ and proceeds in consecutive bands until x is found. In each band, the search procedure is the same as that defined for classical skip lists [Pugh 1990], however, if x is not found in a band, the last visited item is used as a finger into the band below to continue searching. For example, see the search for item 25 in Figure 1 (the reader familiar with *fractional* *cascading* [Chazelle and Guibas 1986] should find some similarities here). Assume that x resides in band $\mathcal{B}_{j>1}$. After retrieving x, this item is promoted to $\mathcal{B}_1$ by extending its column of $H_1 - H_j = O(2^j)$ elements. The procedure RANDOM SELECT is subsequently executed on the bands $\mathcal{B}_i$, $i = 1, 2, \ldots, j-1$. For each $\mathcal{B}_i$, one item residing in the band is demoted to $\mathcal{B}_{i+1}$ in order to keep constant the number of items residing in each band.

Recall that each item's column consists of a deterministic part, used as an anchor point to the skip list below, and a randomized part that plays the same "balancing" role as in a classical skip list. It is crucial to observe that the search process traverses only random parts of the columns, since a column is met for the first time at its topmost element. Then searching for an item x in a band $\mathcal{B}_i$ takes $O(\log_2(\alpha_i + \beta_i)) = O(2^i)$ expected time. Since the search proceeds from the topmost band $\mathcal{B}_1$ to the band actually containing x, say $\mathcal{B}_j$, the overall expected time for searching is $O(\sum_{i=1}^{j} 2^i) = O(2^j)$. Notice that the doubly-exponential increase in the size of SASL bands allows us to make the overall search cost proportional only to the size of the last visited band, $\mathcal{B}_j$. A restructuring cost must be added to this value because of the promotion and cascade demotion steps. However, this does not increase the total cost in order of magnitude. To prove this point, we first make two observations as an immediate consequence of Definition 3.2.

Observation 3.4. Promoting an item x from band $\mathcal{B}_i$ to $\mathcal{B}_1$ requires updating only the counters of x, and the items at the left of x encountered in a skip list search for x.

In the example of Figure 1, promoting item 25 from $\mathcal{B}_3$ to $\mathcal{B}_1$ requires the following changes: The top of column 25 is lifted to $\mathcal{B}_1$. Consider item 35, that is, the only item encountered in the search for 25 and placed at its right. We have $c_1(35) = 1$ (item 35), $c_2(35) = 1$ (item 41), $c_3(35) = 3$ (items 36, 39, and 43). Then, the new counters $c_1(25)$ and $c_2(25)$ are, respectively, set to the values $c_1(35) + 1 = 2$ (item 25 is now counted) and $c_2(35) = 1$. The counter $c_3(25)$ is updated to the value $c_3(25) + c_3(35) - 1 = 6$ (item 25 is counted no more). Consider items 8, 15, 16, and 23 preceding 25 in the search. Items 8 and 15 reside in $\mathcal{B}_1$, thus their counters c_1 are increased by 1 because 25 is brought to their band, and their counters c_3 are decreased by 1 because 25 is no longer in $\mathcal{B}_3$. The counters c_3 of 16 and 23 are instead decreased by the original value of $c_2(25) = 4$ (items 25, 27, 31, and 34) because 25 is no longer in $\mathcal{B}_3$ and column 25 now "hides" items 27, 31, and 34. No counter c_2, except for $c_2(25)$, is changed because in this case, the new column 25 does not hide any element residing in $\mathcal{B}_2$, and no new element is brought to reside in this band. More details will be given in the proof of Theorem 3.6.

Note that after x has been promoted to $\mathcal{B}_1$, this band includes three items instead of two, and the SALS must be rearranged by the demotion process.

Observation 3.5. Demoting an item x from band $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$ requires updating only the counters c_i, c_{i+1} of x, and the items at the left of x encountered in two skip list searches for x, one before and one after the demotion.

In the example of Figure 1, assume that item 41 has to be demoted from $\mathcal{B}_2$ to $\mathcal{B}_3$. The top of column 41 is lowered by four positions. The counter $c_2(41)$ is cancelled. The counter $c_3(41)$ is set to 1 (for 41 itself) plus the number of items in $\mathcal{B}_3$ that can still be reached from 41 (in this case, we have only item 43, before and after demotion). So we set $c_3(43) = 2$. Before demotion, items 8, 15, and 35, at the left of 41, are encountered in the search for this item, so their counters c_2 are decreased by 1, and their counters c_3 are increased by 1. After demotion, a new item 39 is encountered at the left of 41, so its counter c_3 is increased by $c_3(41)$, that is, we set $c_3(39) = 3$.

The counter update operations after promotion or demotion, illustrated in the preceding examples, can be easily formalized algorithmically. This would require introducing some new notation for defining, in each band, the items encountered at the left and right of item x during the search for it. We leave the implementation details to the reader. Still, we are interested in the total amount of work needed for searching and updating the SALS, as evaluated in the following theorem.

THEOREM 3.6. *The search for an item x residing in band $\mathcal{B}_j$ and consequent SASL restructuring takes $O(2^j)$ expected time. The space occupied by the SASL is $O(n \log n)$ expected bits.*

PROOF. We know that retrieving x and identifying its band of residence takes $O(2^j)$ expected time. Additionally, x is then promoted to the band $\mathcal{B}_1$ by extending its column of $H_1 - H_j = O(2^j)$ elements, which takes $O(2^j)$ time. Subsequently, the procedure RANDOM SELECT is executed on each band $\mathcal{B}_i$, $i = 1, 2, \ldots, j-1$ so as to demote one item per band. From Theorem 3.3 and the description of the demotion process given in the previous subsection, the expected time needed to demote all these items is $\sum_{i=1}^{j-1} O(2^i) = O(2^j)$. Furthermore, the values of the counters c_h must be updated, as indicated in Observations 3.4 and 3.5, and this cost must be added.

Let $x_1, \ldots, x_s$ and $y_1, \ldots, y_r$ be the items encountered in the skip list search for x, respectively located at the left and right of x. We have $s \geq r$ and $s = O(2^j)$. Promoting x to $\mathcal{B}_1$ requires creating new counters $c_i(x)$, $1 \leq i \leq j-1$ and updating $c_j(x)$, as well as updating all the existing counters c_i of $x_1, \ldots, x_s$, $1 \leq i \leq j$. Since column x may now hide several items at its right, the values by which the aforementioned counters are changed depend in general on the values of the counters of $y_1, \ldots, y_r$, which give the number of items now hidden by column x. Note that for an arbitrary item z, each counter $c_i(z)$ is represented with $O(2^i)$ bits. Therefore, all the counters $c_i(z)$, $1 \leq i \leq j$ may be stored in a unique word of $O(2^j) = O(\log n)$ bits, and updated in constant time with a unique word addition. The total time for counter updating after promotion is thus $O(s) = O(2^j)$.

For the demotion of x from band $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$, let $x_1, \ldots, x_r$ be the items at the left of x encountered in the skip list search for x before demotion, and let $x_{r+1}, \ldots, x_s$ be further items encounterd in the search after demotion. Note that $s = O(2^{i+1})$. Demoting x requires decreasing $c_i(x_k)$ by 1, and increasing $c_{i+1}(x_k)$ by 1, for $1 \leq k \leq r$, which cancels $c_i(x)$, updating $c_{i+1}(x)$ and $c_i(x_k)$, $c_{i+1}(x_k)$, for $r+1 \leq k \leq s$, by values that depend on counters of the elements encounterd at the right of x. The latter updatings are performed in an "inverse" fashion, as done for promotion. The time for one demotion is then $O(2^{i+1})$, hence the total time for the $j-1$ demotions is $O(\sum_{i=1}^{j-1} 2^{i+1}) = O(2^j)$.

Furthermore, as the changes of all counters of an item s are known at the time that the search passes through s, such changes can be done immediately.

For evaluating the space occupancy, we have seen that $O(n \log n)$ bits are needed for storing the c_h values. The space needed by random parts of the items' columns is $O(n \log n)$ expected bits [Pugh 1990], and it can be immediately seen that the same bound holds for deterministic parts. $\square$

Since the bands of residence of these several items change as search operations are executed, for estimating the overall cost of a sequence of searches, we need to compute the expected value of the index of the band containing each searched for item. Although a frequently accessed item may unfortunately be demoted at random to a low band, the probability that this happens decreases exponentially as we move from higher to lower bands because of their doubly-exponential increase in size. Therefore, on average, the position of an item will be that which it would have occupied if the demotion procedure had been driven by an LRU-strategy (least recently used strategy) applied to every band. This eventually justifies our two choices: bands of doubly-exponential size and random demotion. In fact, we have:

THEOREM 3.7. *In a SASL built over n items, a sequence $\mathcal{Q}$ of m searches requires $O(\sum_{i=1}^{n} n_i \log \frac{m}{n_i})$ expected time, where n_i is the number of times the ith item is accessed in the sequence.*

PROOF. We essentially follow the proof of Theorem 3.3 in Martel [1991], with a further difficulty that the number of items appearing in a band is now a random variable with constant expectation (see Theorem 3.1). Consider an item s, and denote by T the number of operations executed since the last lookup to s in $\mathcal{Q}$ (or, alternatively T is the number of operations executed from the beginning of $\mathcal{Q}$ if s is searched for the first time: In this case, we assume that s has been originally inserted in $\mathcal{B}_1$—see the following). Of course, T is an upper bound to the number of *distinct* items accessed in $\mathcal{Q}$ until this instant.

The main idea is to show the probability that s resides in a band $\mathcal{B}_{k+j}$ after T demotions falls off doubly exponentially in j. The probability, p_{k+j}, that s is demoted from $\mathcal{B}_{k+j}$ to $\mathcal{B}_{k+j+1}$ is 1 minus the probability that all T demotions operated on $\mathcal{B}_{k+j}$ would miss s. Since the number of items residing in $\mathcal{B}_{k+j}$ and possibly affected by the random demotions is $B_{k+j} = 2^{2^{k+j-1}}$, we have: $p_{k+j} = 1 - (1 - \frac{1}{B_{k+j}})^T$.

Let $k = \log \log T$. The probability that after T accesses, item s resides in $\mathcal{B}_{k+j}$ is less than the probability that in these T accesses, s was in $\mathcal{B}_{k+j}$ and has been never demoted from there. In other words:

$$\mathbb{P}[s \text{ resides in } \mathcal{B}_{k+j} \text{ after } T \text{ accesses}] \leq 1 - p_{k+j} = \left(1 - \frac{1}{B_{k+j}}\right)^T \leq e^{-T/B_{k+j}}.$$

The expected cost for looking up item s after T accesses to different items is then from Theorem 3.6:

$$\sum_{j=0}^{b(n)-k} \mathbb{E}[\text{time to look-up} s \mid s \text{ resides in} \mathcal{B}_{k+j}] \times \mathbb{P}[s \text{ resides in} \mathcal{B}_{k+j} \text{ after} T \text{accesses}] =$$

$$= O\left(\sum_{j=0}^{+\infty} 2^{k+j}\, e^{-T/B_{k+j}}\right) = O(2^k) = O(\log T).$$

Let us now consider the overall sequence of lookups to an item s, and denote by $T_1, \ldots, T_{n_s}$ the number of operations between two consecutive lookups to this item. Because of the concavity of the logarithmic function, the summation $\sum_{j=1}^{n_s} O(\log T_j)$ is maximized when $T_j = m/n_s$ for all j. Therefore, the expected cost to access n_s times item s in a sequence of m operations is $O(n_s \log \frac{m}{n_s})$, and summing up on all items, we get the stated bound. $\square$

The operations of insertion and deletion are immediate. Insertion of an item s is always performed in the topmost band by setting $d(s) = H_1$ and flipping a coin to compute $r(s)$. Since the number of items residing in each band is fixed, the insertion triggers a cascade of demotions which eventually ends into the bottom band, thus requiring overall $O(\log n)$ expected time. Updating the counters is done as if s had been in the bottommost band and was then promoted to $\mathcal{B}_1$.

Deletion of an item s consists in removing the column of s and updating consistently all the affected lists and counters, which can be done concurrently with the search for s. If s was residing in band $\mathcal{B}_j$, a promotion from band $\mathcal{B}_{j+1}$ must be done by taking a random item from it with consequent restructuring of the SASL. The cost of a deletion is then $O(\log n)$ expected time.

Since insertions and deletions change the number of items in the lowest band, this can imply a transformation of the SASL structure (see Section 3.1). In fact, if an insertion is done in a SASL $\mathcal{L}$ with a nonempty portion $\mathcal{L}_{down}$ containing $\frac{2}{3}2^{2^{b(n)-1}}$ items, a deterministic column part for each item in the portion $\mathcal{L}_{up}$ of $\mathcal{L}$ is added to the band $\mathcal{B}_{b(n)}$ in $\mathcal{L}_{down}$, which then becomes the last band of $\mathcal{L}_{up}$, while $\mathcal{L}_{down}$ becomes empty. The time for this construction is amortized by the $\Theta(2^{2^{b(n)-1}})$ insertions performed before this. Analogously, if an item is deleted when $\mathcal{L}_{down}$ is empty and $\mathcal{B}_{b(n)}$ contains $\frac{1}{3}2^{2^{b(n)-1}}$ elements, then this band is transferred in $\mathcal{L}_{down}$ after deleting all its deterministic columns. Again, this operation is amortized by the previously performed $\Theta(2^{2^{b(n)-1}})$ deletions.

4. *SASL in External Memory and the Treatment of Strings*

The structure of SASL can be adapted for managing *arbitrary long strings* residing in external memory, where the algorithmic cost is measured in terms of number of disk accesses (I/Os) fetching pages of a fixed size B.

For adapting a SASL to the external memory model, we can simply set the probability of producing a head to $\Theta(1/B)$, as done in Callahan et al. [1995] for B-trees, so that the height of the skip list will be $O(\log_B n)$. Our SASL algorithms work efficiently in this new context due to the randomized demotion strategy and the avoidance of any crosspointer: The crucial observation is that restructuring is done in a downward traversal so that horizontal and vertical pointers (and disk pages) can be updated by exploiting locality of reference.

The real problem is adapting a SASL $\mathcal{L}$ for managing *string items* of arbitrary length. Since an item may occur in $\mathcal{L}$ many times in its column, we substitute these occurrences with pointers to the actual string residing on disk. This introduces one level of indirection, thus requiring accessing the disk anytime two strings have to be compared. Nonetheless, we show that the cost of string comparison can be amortized by accessing the disk only when this is absolutely needed.

The key notion is the one of *longest common prefix* between two arbitrary strings S' and S'', particularly its length (i.e., number of characters), henceforth denoted by $lcp(S', S'')$. In Grossi and Italiano [1999], the notion of lcp has been used to adapt pointer-based data structures to the management of strings. Here, we follow a simpler approach driven by the structural properties of skip lists. These will lead us to prove our Static Optimality theorem, which we regard as our main result.

In a list $\mathcal{L}_i$ of our SASL, consider an element s pointing to a string S. Let $pred(S)$ and $succ(S)$ be strings stored in the predecessor and successor of s in $\mathcal{L}_i$, respectively. Augment the data structure by storing the value $lcp(S, succ(S))$ together with s, and by making the lists $\mathcal{L}_i$ doubly linked via the (horizontal) pointers *succ* and *pred*. Let P be the string to be searched for in a collection of n strings. We shall use a function $extend\_lcp(l, P, \alpha)$ that computes the *lcp* between the two strings P and α, knowing that they have at least l initial characters in common. In other words, the function starts comparing the two strings from their $(l + 1)$th character. We now show how to perform the basic step of the search operation in a skip list, after which what remains to do is just apply the operation repeatedly until the string P is found.

Searching for a string $P[1, p]$ in $\mathcal{L}$ proceeds per levels by maintaining an invariant on the parameters L_i, R_i, M_i so that

—L_1 and R_1 are the special strings $-\infty$ and $+\infty$, respectively, and M_1 is a string of $\mathcal{L}_1$ which shares the longest prefix with P (recall that $|\mathcal{L}_1| = O(1)$);

—for $i > 1$, L_i and R_i are the two strings delimiting the portion of $\mathcal{L}_i$ where the search has been confined by the previous steps on $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$; and

—for $i > 1$, M_i is the string of $\mathcal{L}_i$ between L_i and R_i which shares the longest prefix with P. We also maintain the value $l_i = lcp(M_i, P)$.

The search in $\mathcal{L}_i$ can go forward or backward over items between L_i and R_i, depending on the lcp-value among M_i and P (note the difference with Ferragina and Grossi [1999]). I/Os are possibly needed for comparing P with the strings between L_i and R_i, however, these I/Os gather new information on the longest prefix shared by P, which can be saved in subsequent comparisons.

As usual in skip lists, we check whether $succ(L_i) = R_i$, and in this case the search proceeds into the next list $\mathcal{L}_{i+1}$ by setting $L_{i+1} = L_i$, $R_{i+1} = R_i$, and $M_{i+1} = M_i$. Assume instead that $succ(L_i) \neq R_i$. Unlike classical skip lists, the next item x to be compared against P is chosen between $succ(L_i)$ and $pred(R_i)$, depending on the current value of M_i. Indeed, we know the value $l_i = lcp(M_i, P)$ by the inductive search and thus take $x = succ(L_i)$ if $M_i = L_i$, and $x = pred(R_i)$ otherwise. The value $l' = lcp(M_i, x)$ is available in the data structure, and we can get this in $O(1)$ I/Os. Essentially, we use l' to drive the next step of the search, as follows:

—If $l' < l_i$ then: if $M_i = L_i$ then set $R_i = x$ else set $L_i = x$ (in both cases M_i and l_i remain unchanged);

—if $l' > l_i$ then: if $M_i = L_i$ then set $L_i = x$ else set $R_i = x$ (in both cases M_i and l_i remain unchanged);

—if $l' = l_i$ then

—$l_i = extend\_lcp(l_i, P, x)$; $M_i = x$;

—if $P[l_i] < M_i[l_i]$ then $R_i = M_i$ else $L_i = M_i$.

The key point here is that, as in other data structures [Manber and Myers 1993; Ferguson 1992; Ferragina and Grossi 1999], only one string is compared at each disk page traversed during the forward or backward scan over items in $[L_i, R_i]$, and this comparison rescans merely a constant number of pattern characters. This gives a telescopic sum on the number of I/Os bounded above by $O(\log_B \beta_j + |P|/B)$, where β_j is the number of items residing in $\mathcal{B}_j$, the band containing the searched string.

All other computations involve local information and thus can be performed within the same I/O-bounds of SASL in external memory. Updating the data structure as a consequence of an insertion, deletion, or restructuring operation then becomes more elaborate, but follows the ideas detailed for the internal memory case, and thus, we leave the details to the reader. We just note that by exploiting the lexicographic ordering of the strings in $\mathcal{L}$, the $lcp()$ information can be updated during the downward traversal of the list, together with the update of the bidirectional pointers.

From the preceding arguments, we immediately have:

THEOREM 4.1 (STATIC OPTIMALITY THEOREM). *In an external-memory SASL built over n strings $S_1, \ldots, S_n$ of total length N, a sequence of m string searches $S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ requires $O(\sum_{j=1}^{m}(\frac{|S_{i_j}|}{B}) + \sum_{i=1}^{n}(n_i \log_B \frac{m}{n_i}))$ expected I/Os, where n_i is the number of times the ith string S_i is queried. Inserting or deleting a string S takes $O(\frac{|S|}{B} + \log_B n)$ expected I/Os. The total space occupied by the data structure is $O(N/B)$ expected disk pages.*

5. *Caching Into the Internal Memory*

The I/O-bound provided in Theorem 4.1 is based on the self-adjusting discipline, and the way this copes with the frequency of string accesses in the query sequence. The string nature of the items is not exploited and, in fact, strings are treated as if they were atomic items by moving their pointers around. Then, if two items have a portion (typically, a prefix) in common, this is actually scanned twice. Usually, this repetitiveness is exploited by the *buffering/caching* policies of the underlying operating system. However, we wish to extend our data structure to explicitly take this into account. In particular, we propose that the internal memory be used persistently across queries so that it behaves as a cache, and define caching policies integrated into the SASL.

Assume that a *cache* $\mathcal{C}$ of size c disk pages is established in internal memory, containing a copy of some prefixes of queried strings whose length is a multiple of B. Let $S[1, iB]$ be a cached prefix. For each shorter prefix $S[1, jB]$, with $1 \leq j \leq i$, keep a pointer to the *topmost* string having this prefix, that is, to the first string encountered in the SASL when searching for a string prefixed by $S[1, jB]$. These pointers are used as "shortcuts" into the SASL for speeding-up the search when a prefix of the queried string is available in the cache.

Let P be the currently queried string. Search in $\mathcal{C}$ for the longest cached prefix $P[1, iB]$, and then follow the pointer associated to this prefix so as to reach the topmost string in the SASL prefixed by $P[1, iB]$. Then continue with a standard search in the SASL for the remaining suffix $P[iB + 1, |P|]$. This amounts to saving i I/Os in the search process, specifically those due to scanning the first i pages of P. We have then to estimate the average value of i as a function of the "regularity" (i.e., the entropy) of the query sequence, and of the size of $\mathcal{C}$.

Before proceeding with these calculations, however, we briefly indicate some major implementation rules. After P has been queried, the cached prefix $P[1, iB]$

is extended by loading the next page $P[iB+1, (i+1)B]$ in $\mathcal{C}$. Then, the pointer associated to this longer prefix is set, noting that the pointed string lies in the path followed in the search for $P[iB+1, |P|]$. For loading the new page in $\mathcal{C}$, another page must be evicted, chosen among the oldest final pages of cached prefixes; this way, some cached prefix has one page less. As far as keeping the shortcut pointers under promotion/demotion operations and insertion/deletion of a string in the SASL, we limit ourselves to explaining the demotion case, as the other cases are similar. The demoted string S might be the destination of some pointers that, therefore, have to be modified. However, these pointers may be only moved from S to its adjacent string to the right, in the same level of the skip list. We can test if this is the case by exploiting knowledge on the longest common prefix information kept between adjacent strings on the same level, hence, each pointer can be moved together with the demotion of S, without requiring any further I/Os.

The following theorem measures the goodness of our caching strategy as a function of cache size c and entropy $\mathcal{H}$ of string prefixes of a given length rB in the query sequence (recall that for a set of n items with probabilities $p_1, \ldots, p_n$, we have $\mathcal{H} = -\sum_i p_i \log_2 p_i$). For this purpose, we introduce a parameter γ accounting for the average number of cache misses for each queried string. We are also assuming that the strings have length smaller than the cache size, and shall comment on the general case after the theorem.

THEOREM 5.1. *Let $S_1, \ldots, S_n$ be a set of n strings having total length N, and $\mathcal{Q} = S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ be a query sequence of length m. The combination of SASL with a cache $\mathcal{C}$ of size cB (i.e., equal to c disk pages) allows answering $\mathcal{Q}$ in*

$$O\left(\sum_{j=1}^{m}(\gamma(S_{i_j})) \;+\; \sum_{i=1}^{n}\left(n_i \log_B \frac{m}{n_i}\right)\right)$$

amortized expected I/Os, where $\gamma(S_{i_j}) = \min\{\frac{|S_{i_j}|}{B}, \sum_{r=1}^{|S_{i_j}|/B} \frac{\mathcal{H}[r]}{\log(c/r)}\}$ measures the average number of cache misses, $\mathcal{H}[r]$ is the entropy of string prefixes of length rB, and n_i is the number of times the string S_i is queried in $\mathcal{Q}$. Inserting or deleting a string S takes $O(\frac{|S|}{B} + \log_B n)$ amortized expected I/Os. The average space occupied by the data structure is $O(N/B)$ disk pages.

PROOF. Instead of evaluating the behavior of our caching strategy directly, we consider a more complicated approach which is simpler to analyze. In fact, we assume that a page $S[iB+1, (i+1)B]$ is loaded in $\mathcal{C}$ only if the prefix $S[1, (i+1)B]$ has been queried at least $(i+1)$ times during the last c string queries. The probability of a cache hit in this new approach is smaller than that in our caching strategy. Then: evaluating the probability of a cache miss gives an upper bound for the approach actually adopted. Let:

—$A = \sqrt{i/c} > (i/c)$, since $i < c$;

—$p_i = \sum_{\substack{\text{all string prefixes } \sigma \\ |\sigma| = iB,\; p(\sigma) < A}} p(\sigma)$; and

—$\mathcal{H}[i]$ = entropy of prefixes of length iB over the query sequence.

The quantities p_i and $\mathcal{H}[i]$ are related as follows:

$$\mathcal{H}[i] = \sum_{\substack{\text{string prefixes } \sigma \\ |\sigma|=iB,\ \mathbb{P}(\sigma)<A}} \mathbb{P}(\sigma)\log(1/\mathbb{P}(\sigma)) + \sum_{\substack{\text{string prefixes } \sigma \\ |\sigma|=iB,\ \mathbb{P}(\sigma)\geq A}} \mathbb{P}(\sigma)\log(1/\mathbb{P}(\sigma)) \geq$$
$$\geq \sum_{\substack{\text{string prefixes } \sigma \\ |\sigma|=iB,\ \mathbb{P}(\sigma)<A}} \mathbb{P}(\sigma)\log(1/A) \geq p_i \log(1/A)$$

Hence, we have

$$p_i \leq \frac{\mathcal{H}[i]}{\log(1/A)} = \frac{2\mathcal{H}[i]}{\log(c/i)}.$$

For estimating the probability that the ith page of a query string S is not in the cache (i.e., the prefix $S[1, iB]$ of this string occurred, at most, $(i-1)$ times as a prefix of the previous c queried strings), we introduce a random variable X_σ^c which counts the number of strings having prefix σ, with $|\sigma| = iB$ in a sequence of c queried strings.The value of X_σ^c can be computed as a sum of Bernoulli variables, defined as:

$$X_\sigma^c[j] = \begin{cases} 1 & \text{if the } j\text{th string in a window of } c \text{ query strings has prefix } \sigma, \\ 0 & \text{otherwise.} \end{cases}$$

Let $X_\sigma^c = \sum_{j=1}^{c} X_\sigma^c[j]$, and observe that $\mathbb{E}[X_\sigma^c] = \mathbb{P}(\sigma)c$. We can now compute the probability that the ith page of a queried string incurs in a cache miss, that is, the cached prefix is shorter than iB. We have

$$\begin{aligned}
&\mathbb{P}(\text{cache miss on the } i\text{th page}) \\
&\quad = \sum_{\sigma:\,|\sigma|=iB} \mathbb{P}(\text{cache miss on } i\text{th page}|\text{query prefix is } \sigma)\ \mathbb{P}(\sigma) \\
&\quad = \sum_{\sigma:\,|\sigma|=iB} \mathbb{P}(X_\sigma^c < i)\cdot\mathbb{P}(\sigma) \\
&\quad = \sum_{\substack{\sigma:\,|\sigma|=iB \\ \mathbb{P}(\sigma)<A}} \mathbb{P}(X_\sigma^c < i)\ \mathbb{P}(\sigma) + \sum_{\substack{\sigma:\,|\sigma|=iB \\ \mathbb{P}(\sigma)\geq A}} \mathbb{P}(X_\sigma^c < i)\ \mathbb{P}(\sigma), \qquad (1)
\end{aligned}$$

where the first term can be bounded as

$$\sum_{\substack{\sigma:\,|\sigma|=iB \\ \mathbb{P}(\sigma)<A}} \mathbb{P}(\sigma) = p_i \leq \frac{2\mathcal{H}[i]}{\log(c/i)}. \qquad (2)$$

For finding an upper bound to the second term, recall that the Chernoff bound for a sum of independent Bernoulli variables X_i, having average $\mathbb{E}[\sum X_i] = \mu$, is given by $\mathbb{P}(\sum X_i < (1-\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}$. Using this bound with $\delta = 1 - \frac{i}{\mathbb{P}(\sigma)c}$ (so that $(1-\delta)\mathbb{E}[X_\sigma^c] = i$), we have:

$$e^{-\frac{\mu\delta^2}{2}} = e^{-\frac{\mathbb{P}(\sigma)c}{2}(1-\frac{i}{\mathbb{P}(\sigma)c})^2}.$$

Since $\mathbb{P}(\sigma) > A > \frac{i}{c}$, we correctly have $\delta < 1$ and $\delta > 0$. Moreover, $c\,\mathbb{P}(\sigma) > cA = \sqrt{c\,i}$ and $\frac{i}{\mathbb{P}(\sigma)c} < \frac{i}{Ac} = \sqrt{\frac{i}{c}}$, hence:

$$e^{-\frac{\mathbb{P}(\sigma)c}{2}(1-\frac{i}{\mathbb{P}(\sigma)c})^2} \leq e^{-\frac{\sqrt{ic}}{2}(1-\sqrt{\frac{i}{c}})^2}.$$

Assuming that $i \leq \alpha c$ with $\alpha < 1$, as we refer to caching of prefixes whose length is (at most) a constant fraction of the cache size, we can further bound the previous inequality as follows:

$$e^{-\frac{\sqrt{i\,c}}{2}(1-\sqrt{\frac{i}{c}})^2} \leq e^{-\frac{\sqrt{i\,c}}{2}(1-\sqrt{\alpha})^2} = e^{-\frac{\sqrt{i\,c}}{2}\beta}, \tag{3}$$

where $\beta = (1-\sqrt{\alpha})^2$. Substituting the bounds of relations 2, 3 in Eq. (1) we have

$$\mathbb{P}(\text{cache miss on the } i\text{th page}) \leq \frac{2\mathcal{H}[i]}{\log(c/i)} + e^{-\frac{\sqrt{i\,c}}{2}\beta}$$

from which an upper bound to the expected number of cache misses derives. We have

$$\mathbb{E}[\text{cache misses}] \leq \sum_{r=1}^{\alpha c} \mathbb{P}(\text{cache miss on the } r\text{th page}) < ce^{-\frac{\sqrt{c}}{2}\beta} + 2\sum_{r=1}^{\alpha c} \frac{\mathcal{H}[r]}{\log(c/r)}.$$

For large c, the first term is strictly smaller than 1, which we drop when computing in order of magnitude. The formula captures the repetitiveness at prefix lengths up to αcB. Since by hypothesis, all the strings are shorter than the cache size, the parameter $\gamma(S_{i_j}) = \min\{\frac{|S_{i_j}|}{B}, \sum_{r=1}^{|S_{i_j}|/B} \frac{\mathcal{H}[r]}{\log(c/r)}\}$ captures the number of cache misses incurred by the search for S_{i_j}. The number of I/Os needed for string insertion/deletion, and the size of the data structure, are derived as in Theorem 4.1. $\square$

Note that if the query sequence $\mathcal{Q}$ is repetitive, then the entropy $\mathcal{H}[r] \approx 0$ for all r. In such a case, Theorem 5.1 shows that the whole query string S_{i_j} is cached and no I/Os are incurred by scanning the string. I/Os for SASL rebalancing are still needed and accounted for in the aforementioned second term. A possible improvement is briefly discussed in the next section.

For a very long string S_{i_j}, we pay I/Os for its pages in the suffix $S_{i_j}[\alpha|\mathcal{C}|+1, |S_{i_j}|]$ with $\alpha < 1$, while the cost of accessing the prefix $S_{i_j}[1, \alpha\,|\mathcal{C}|]$ is still given by Theorem 5.1.

6. *Concluding Remarks*

The purpose of this work was studying data structures for external-memory string access under a sequence of string queries. To this end, we have designed a novel self-adjusting skip list (SASL) that adapts itself to the distribution of queries, without explicitly setting weight or structural constraints. SASL managing has led to a Static Optimality theorem for external-memory string accesses, a result that was *a fortiori* left open in Sleator and Tarjan [1985], thus showing that the new data structure is optimal with respect to the best offline static string index. We further improved on the performance by introducing a paradigm under which part of the internal memory is persistently used as a cache across the queries for decreasing the number of disk page accesses. Performance analysis crucially depends on the entropy of string prefixes in the queries. This changes the "nature" of the Static Optimality theorem for string data in the presence of a memory hierarchy.

Many interesting questions remain. First, we leave open the problem of providing a "cache-oblivious" variant for SASL, namely, one that works without any knowledge of the parameters of the memory hierarchy such, as B Frigo et al. [1999]. Second, there are also other forms of self-adjusting structures and results that may be of interest in our context, such as the Working Set theorem [Sleator and Tarjan 1985; Iacono 2001], dynamic optimality [Blum et al. 2003; Sleator and Tarjan 1985], and reference locality [Ergun et al. 2001].

The string dictionary problem at large is not limited to searching string patterns exactly, but may also require answering *prefix-match queries* over stored strings. It goes without saying that SASL also supports this kind of search, and actually, prefixes frequently occurring in the query sequence will actually live into the higher bands of the SASL, thus allowing their faster retrieval. This property is crucial for our third suggested direction of research. Indeed, SASL might be adapted to index *all the suffixes of a given string* S by storing pointers to all of them into $\mathcal{L}$. In this way, SASL would be able to support substring searches over S and maintain a form of self-adjustability among the suffixes of this string by promoting those whose prefixes are more frequently queried. This property, in our opinion, might turn out to be crucial in the context of constructing the SASL by repeated suffix insertions. Recent articles have investigated some forms of suffix bucketing to speed-up the practical construction of large suffix trees [Hunt et al. 2002; Giegerich et al. 2003; Bedathur and Haritsa 2004; Tata et al. 2004; Cheung et al. 2005] by copying prefixes of them into internal memory. The approach based on SASL would possibly provide a technique to choose adaptively the best subset of suffixes to be kept into internal memory for reducing the number of executed I/Os, based on the substring repetitiveness in S.

Finally, some open problems remain concerning the use of internal memory as a cache. In Vitter and Krishnan [1996], optimal prefetching algorithms deriving from compression have been presented and analyzed. The approach adopted there could probably be extended to achieve tighter I/O-bounds. A natural extension of our work would also be to cache the top $\Theta(\log_B M)$ levels of the external-memory SASL, where M is the internal-memory size. In fact, caching the top part of the data structure, in addition to caching the string prefixes as shown in Section 5, clearly improves performance if a query sequence is highly repetitive. This has been investigated experimentally in the context of web search engines [Xie and O'Hallaran 2002].

REFERENCES

ABOULNAGA, A., ALAMELDEEN, A. R., AND NAUGHTON, J. F. 2001. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 591–600.

ABRAMSON, N. 1983. *Information Theory and Coding*. McGraw-Hill, New York.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*. 266–277.

BEDATHUR, S. J., AND HARITSA, J. R. 2004. Engineering a fast online persistent suffix tree construction. In *Proceedings of the 20th International Conference on Data Engineering*. 720–731.

BLUM, A., CHAWLA, S., AND KALAI, A. 2003. Static optimality and dynamic search optimality in lists and trees. *Algorithmica 36*, 3, 249–260.

CALLAHAN, P. B., GOODRICH, M. T., AND RAMAIYER, K. 1995. Topology B-trees and their applications. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, vol. 955, Springer Verlag. 381–392.

CHAZELLE, B., AND GUIBAS, L. J. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica 1*, 2, 133–162.

CHEUNG, C., YU, J., AND LU, H. 2005. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Trans. Knowl. Data Engi. 17*, 1, 90–105.

COOPER, B., SAMPLE, N., FRANKLIN, M. J., HJALTASON, G. R., AND SHADMON, M. 2001. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 341–350.

ERGUN, F., SAHINALP, S. C., SHARP, J., AND SINHA, R. K. 2001. Biased dictionaries with fast insert/deletes. In *Proceedings of the 33rd ACM Symposium on the Theory of Computing*. 483–491.

FERGUSON, D. E. 1992. Bit-Tree: A data structure for fast file processing. *Commun. ACM 35*, 6, 114–120.

FERRAGINA, P. 2005. String search in external memory: Algorithms and data structures. In *Handbook of Computational Molecular Biology*, S. Aluru, ed., Chapman and Hall, London.

FERRAGINA, P., AND GROSSI, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM 46*, 2, 236–280.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-Oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on the Foundations of Computer Science*. 285–298.

GIEGERICH, R., KURTZ, S., AND STOYE, J. 2003. Efficient implementation of lazy suffix trees. *Softw. Pract. Exper. 33*, 1035–1049.

GROSSI, R., AND ITALIANO, G. 1999. Efficient techniques for maintaining multidimensional keys in linked data structures. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 1644. Springer Verlag. 372–381.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York.

HUNT, E., ATKINSON, M., AND IRVING, R. 2002. Database indexing for large DNA and protein sequence collections. *Int. J. Very Large Data Bases 11*, 3, 256–271.

IACONO, J. 2001. Alternatives to splay trees with $o(\log n)$ worst-case times. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*. 516–522.

KÄRKKÄINEN, J., AND RAO, S. 2003. Full-Text indexes in external memory. In *Algorithms for Memory Hierarchies*, U. Meyer et al. eds. Lecture Notes in Computer Science, vol. 2625. Springer Verlag, 149–170.

MANBER, U., AND MYERS, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput. 22*, 5, 935–948.

MARTEL, C. 1991. Self-Adjusting multi-way search trees. *Inf. Process. Lett. 38*, 3, 135–141.

MEHLHORN, K. 1984. *Data Structures and Algorithms 1: Searching and Sorting*. EATCS Monographs on Theoretical Computer Science, vol. 1. Springer Verlag.

MEHLHORN, K., AND NAHER, S. 1992. Algorithm design and software libraries: Recent developments in the Leda project. In *IFIP World Computer Congress*, vol. 1, 493–505.

MULMULEY, K. 1994. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, Upper Saddle River, NJ.

PUGH, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33*, 6, 668–676.

SHERK, M. 1995. Self-Adjusting k-ary search trees. *J. Algorithms 19*, 1, 25–44.

SLEATOR, D., AND TARJAN, R. 1985. Self-Adjusting binary search trees. *J. ACM 32*, 3, 652–686.

TATA, S., HANKINS, R. A., AND PATEL, J. M. 2004. Practical suffix tree construction. In *Proceedings of the 13th International Conference on Very Large Data Bases*. 36–47.

VITTER, J. 2002. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Comput. Surv. 33*, 2, 209–271.

VITTER, J. S., AND KRISHNAN, P. 1996. Optimal prefetching via data compression. *J. ACM 43*, 5, 771–793.

XIE, Y., AND O'HALLARAN, D. 2002. Locality in search engine queries and its implications for caching. In *Proceedings of the 21st IEEE INFOCOM Conference*. 1238–1247.

RECEIVED NOVEMBER 2004; ACCEPTED SEPTEMBER 2006