Università di Pisa
Dipartimento di Informatica

Technical Report: TR-01-26

Self-adjusting Data Structures for External Memory String Access

V. Ciriani
University of Pisa

P. Ferragina
University of Pisa

F. Luccio
University of Pisa

S. Muthukrishnan
AT&T Labs – Research

November 20, 2001

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Self-adjusting Data Structures for External Memory String Access

V. Ciriani
University of Pisa

P. Ferragina
University of Pisa

F. Luccio
University of Pisa

S. Muthukrishnan
AT&T Labs – Research

November 20, 2001

Abstract

Data warehouses are increasingly storing and managing large scale string data, and dealing with large volume of transactions that update and search string databases. Motivated by this context, we initiate the study of *self-adjusting* data structures for string dictionary operations, that is, data structures that are designed to be efficient on an entire sequence of operations rather than individual string operations. Furthermore, we study this problem in the external memory model where string data is too massive to be stored in main memory and has to reside in disks; each access to a disk page fetches B items, and the cost of the operations is the number of pages accessed.

We show that given n strings $S_1, \ldots, S_n$ of total length $\sum_i |S_i| = N$, a sequence of m string searches $S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ takes $O(\sum_{j=1}^{m}(\frac{|S_{i_j}|}{B}) + \sum_{i=1}^{n}(n_i \log_B \frac{m}{n_i}))$ amortized expected I/Os, where n_i is the number of times S_i is queried. Inserting or deleting a string S takes $O(\frac{|S|}{B} + \log_B n)$ amortized expected I/Os. This result is the analog of what is known as the Static Optimality Theorem [17] proved by Sleator and Tarjan in their classic splay trees paper for a dictionary of numerical values; here, it has been generalized, for the first time to string data, to string operations, and to the external memory model. This performance is achieved not by traditional "splay" operations on search trees as in [17], but by designing a novel self-adjusting data structure based on the well-known skip lists.

In addition, we introduce the paradigm of using the main memory (or a part thereof) persistently across operations, in the manner of a cache, to further improve the performance of our self-adjusting skip list. This is quite reasonable in applications where string operations are large in volume. The resulting bound on the complexity of sequences of string operations in external memory achieved by our self-adjusting skip list is related to the "entropy" of *string prefixes* in the query sequence; this is a novel aspect that arises in the nature of the Static Optimality Theorem for external memory string accesses.

1 Introduction

Conventional data warehouses such as product inventories and customer databases that primarily manage numerical data or fixed length keys are getting augmented by emerging datastores that archive string data which are alphanumeric, significantly varying in length, and far more massive. Sources of string data are many: patent databases that contain full texts of patent documents, online libraries, biological databases, XML databases, product catalogs, webserver logs and other Internet traffic data, customer feedback and annotations, etc. Not only is string data proliferating, but datastores increasingly handle large number of *string transactions* that add, delete, modify or search strings. As a result, the problem of managing massive string data under large number of transactions is emerging as a challenge.

We study the fundamental problem of maintaining a string dictionary, that is, support insertion and deletion of strings and search for patterns. Traditionally, string algorithms focus on supporting each of these operations *individually* in the most efficient manner. However, due to the discussion above, we are motivated to consider massive transactions, and seek dictionary solutions that are efficient for the entire stream of transactions, rather than for the worst case per operation. In particular, in the spirit of *amortized analysis* [17], we seek solutions that are competitive over the entire sequence of operations. Note that the worst-case cost over a sequence of operations may be far less than the total worst-case cost per operation summed over the set of all operations in the sequence.

In a classic paper [17], Sleator and Tarjan presented an adaptive data structure for *numeric-valued dictionaries* that automatically reorganizes itself in response to 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*: given m accesses to an n-node splay tree, the total time for the m accesses is $O(m + \sum_{i=1}^{n} q(i) \log(m/q(i)))$ where $q(i)$ is the number of times the ith item occurs in the query sequence. The bound would have been $\Theta(m \log n)$ if we considered the total cost as the sum of worst case costs per operation, but the amortized bound above is better when the sequence of accesses is skewed. The splay tree is in fact as efficient as any fixed search tree since the access cost above 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 [1, 17]; the term $\sum_{i=1}^{n} q(i) \log(m/q(i))$ relates to the entropy of the sequence of accesses. Proving a similar theorem of optimality against dynamic search trees (the so called *Dynamic Optimality Theorem*) still remains open [17].

Designing an amortized solution for the string dictionary problem is challenging in three main respects. **1)** Consider strings as *atomic* items, hash them into numeric space and apply Sleator and Tarjan's work. This does not solve the string dictionary problem because searching for a pattern P in the dictionary $\mathcal{D}$ typically involves determining not merely if P is present in $\mathcal{D}$, but also *partial match queries*, that is, either finding the longest prefix of P that is found in $\mathcal{D}$ (as a prefix or a substring), or finding the lexicographic predecessor of P. No hashing function is known that captures these string relationships. **2)** The *nature* of the static optimality theorem itself remains unclear for string accesses. This is because successive patterns $P_1, P_2, \ldots$ that get searched are not atomic items; they have common relationships. For example, even though "cool glasses" may appear only a few times in the sequence of searches, "cool town", "coolant", and "cool jazz" may appear frequently. Therefore, a notion of optimality over a string sequence has to consider the frequency of prefixes, not necessarily the entire strings. Finally, **3)** due to our motivating discussions above, we would like to provide solutions for string dictionaries that are massive. In particular, it means that the dictionary does not fit into the main memory and large portions of it must reside on external memory, ie., the disk. The standard external memory model has main memory of size M and each access to the disk is in B-sized pages [18]; the cost of operations is the number of I/Os (ie., number of disk page accesses) and not of word operations on main memory items. In existing work on string dictionaries, *lexicographic trees* with splaying [17] address the first challenge, string B-trees address the third, but there are no solutions that address all the three challenges above.

In this paper, we initiate the study of designing amortized data structures for massive string dictionaries and address the challenges above. Our main results are twofold.

1. We prove the first known static optimality theorem for massive string dictionaries. We show that given n strings $S_1, \ldots, S_n$ of total length $\sum_i |S_i| = N$, a sequence of m string searches $S_{i_1}, S_{i_2}, \ldots, S_{i_m}$ requires
$$O\left(\sum_{j=1}^{m} (\frac{|S_{i_j}|}{B}) + \sum_{i=1}^{n} (n_i \log_B \frac{m}{n_i})\right)$$
amortized expected I/Os, where n_i is the number of times S_i is queried. Inserting or deleting a string S takes $O(\frac{|S|}{B} + \log_B n)$ amortized expected I/Os. The total space occupied by the self-adjusting data

structure is $O(N/B)$ expected disk pages.

Clearly $\sum_{j=1}^{m} \frac{|S_{i_j}|}{B}$ disk accesses are needed in the worst case to compare the searched strings against the dictionary strings; the $\sum_{i=1}^{n} n_i \log_B \frac{m}{n_i}$ term is related to the entropy of the query sequence, and is a standard information-theoretic lower bound [11]. The B-way branching provided by the page size of a disk access is fully utilized, that is, $\log_2$ and $|S_i|$ are replaced by $\log_B$ and $|S_i|/B$ respectively. The total bound is analogous to that in Sleator and Tarjan's Static Optimality Theorem, but for the first time, it has been generalized to hold for *external-memory string access*. In contrast to this bound, using string data structures like String B-trees that give the best known performance per operation in the worst case [7], one would have achieved an I/O-bound of $O(\sum_{j=1}^{m}(\frac{|S_{i_j}|}{B}) + m \log_B n)$, which is not optimal with respect to the distribution of the strings in the query sequence.

2. We use the paradigm that main memory can be persistently used across queries much like a cache; as a result, we further strengthen the result above. In particular, with input as above, we show that given a cache $\mathcal{C}$ of size c disk pages, there is a self-adjusting data structure that uses

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

amortized expected I/Os over all m searches, 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 and $\mathcal{H}[r]$ is the entropy of the string prefixes having length rB. The other bounds remain unchanged.

This result is novel in many ways. First, standard results in external-memory data structures [18] exploit the main memory only at construction time; in contrast, we use it persistently across queries, and use it as a cache. This is likely to be a powerful paradigm to exploit in self-adjusting data structures on external memory. Second, although caching strategies of numeric values have been studied quite extensively from 60's, caching strategies for string dictionaries are not well understood. Our work is the first that addresses this issue, presents a simple caching strategy and analyzes it; furthermore, it has been incorporated into our string dictionary data structure. Third, the bound explicitly discounts the number of disk pages accessed for the *repetitive* prefixes in the query. This novel notion of repeating prefixes in queries is formalized in terms of the entropy of "short" string prefixes; and we expect it to find applications elsewhere.

To design competitive string dictionary data structures on external memory, a natual approach is to take an external memory string dictionary data structure and design a "splaying" operation *à la* Sleator and Tarjan [17]. This approach has problems. The only known candidate is the String B-tree introduced in [7] which has structural constraints that maintain its balancedness in presence of string inserts and deletes. Maintaining these constraints under the sophisticated splaying operations known for multiway search trees [16] results in complicated structure and analyses, but ultimately does not provide the Static Optimality Theorem and its generalization above.

We instead design an alternative data structure for our problem. We start with the well-known Skip List proposed for solving the dictionary problem with numerical value in main memory [15]. Skip lists maintain the items of the set in levels, each level consisting of a randomly chosen half of the items from the previous level. It is easy to extend it to work in external memory by choosing items with probability $\Theta(1/B)$ at each level to populate the next level. It is not difficult to extend it to work with strings rather than numbers. The challenge is to make it self-adjusting under these two extensions. In particular, when a string is accessed, we need to *promote* certain items up the levels and simultaneously *demote* certain items down the levels. We present a new data structure that we call *self-adjusting skip lists* (SASL), and prove our results for it. A technical novelty in SASL is a simple randomized demotion strategy that, together with the doubly-exponential grouping of the skip-list levels, guides the demotion; this way frequent items get to remain at the highest levels of the skip lists. Note that our results are randomized (not only due to the basic properties of skip lists, but also due to the randomized demotion strategy of SASL), and hold with high probability; this is a departure from Sleator and Tarjan [17] where the bounds were deterministic.

SASL has many properties that other self-adjusting data structures and skip lists possess. It achieves balancedness without explicit weight and structural constraints on the data structure. SASL is simple and more appealing than the string B-tree which is the data structure of choice for indexing strings in external memory. In particular, our update algorithms are conceptually simpler and easier to implement, need less space and guarantee a better use of disk space because they merely merge lists of disk pages instead of B-tree nodes; in addition, with high probability, our data structure is 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 the time (as most transactions do in practice). On the other hand, as is typical of self-adjusting data structures, individual operations may be expensive, and each access operation needs more local adjustments than in a standard string B-tree.

The rest of the paper is organized as follows. In Section 2, we review standard skip lists and some related work. We present our self-adjusting skip list (SASL) on main memory in Section 3; this is of independent interest. In Section 4, we extend SASL to external memory and derive our first result of the Static Optimality Theorem for external memory string dictionaries. Finally, we describe in Section 5 how to use the main memory as a cache to derive our second main result; in particular, we characterize disk access savings in terms of the entropy of the string prefixes in the query sequence. Concluding remarks are in Section 6; since our paper initiates the study of self-adjusting external memory string dictionaries, several problems remain to be explored.

2 Preliminaries and Related Work

The *external-memory string access* problem takes as input a dictionary $\mathcal{D}$ of n strings $S_1, \ldots, S_n$ of total length N. The problem consists of building an external-memory data structure that indexes the dictionary strings so that a sequence of string searches and updates can be performed efficiently. A search for a string S consists of finding the string in $\mathcal{D}$ sharing the *longest common prefix* with S (or being its lexicographic predecessor/successor). A string update consists of inserting the input string into $\mathcal{D}$ or deleting the input string from $\mathcal{D}$.

Skip lists were introduced by Pugh [15] as a simple alternative to balanced search trees. 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 $\mathcal{S}_0, \ldots, \mathcal{S}_h$ such that: $\mathcal{S}_0 = \emptyset$ and $\mathcal{S}_h = \mathcal{S}$; $\mathcal{S}_{h-i+1}$ is the set of all items for which we flipped the coin at least i times. Notice that $\mathcal{S}_j \subseteq \mathcal{S}_j$. h is a random variable which depends on the outcomes of the coin flips; it can be easily proved that on the average $\mathbb{E}[h] = O(\log n)$ [15]. The *skip list* for $\mathcal{S}$ consists of a collections of lists organized as follows (unlike [15], we number the lists from the smallest to the largest):

- for each $i \geq 1$, the list $\mathcal{L}_i$ is doubly-linked and contains the elements of $\mathcal{S}_i \cup \{-\infty, +\infty\}$ stored in increasing order;
- for each $i \geq 1$, each item in $\mathcal{L}_i$ has a "vertical" pointer to its occurrence in $\mathcal{L}_{i+1}$.

It is easy to prove that the expected size of each list grows exponentially, that is $\mathbb{E}[|\mathcal{L}_i|] = \Theta(2^i)$.

The basic operation is *searching for an item x* in $\mathcal{S}$. It consists of traversing the skip list per increasing levels, starting from $\mathcal{L}_1$ at $-\infty$. Each list $\mathcal{L}_i$ is percolated rightward 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 seach is succesful 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 rightward traversal is repeated. When i equals h, the search is deemed unsuccesful. Search, insert, and delete operations in a skip list can be implemented as described above and take $O(\log n)$ expected time each [15].

The problem with skip lists is that they are not self-adjusting. In fact, 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 $\mathcal{S}$. Self-adjusting skip list should "artificially" promote the frequently accessed items to the highest levels of $\mathcal{L}$ thus facilitating their retrieval. This has been achieved in various ways [6, 14, 13], but either assuming that the frequency of accesses are known or under specific assumptions such as probability of accesses being non-increasing over time.

A recent proposal in this direction has been made by [6] who introduced *biased skip lists* (BSL) as a self-adjusting form of classical skip lists running in the RAM (main memory) model. In [6] each item has associated an *MTF-rank* (Move-to-Front rank) which determines its level in the underlying skip list. Smaller the MTF-rank, higher the level to which the item belongs. BSL supports search/insert/delete operations in (suppressing certain details not pertinent to us here) $O(\log_2 r)$ expected amortized time, where r is the MTF-rank of the queried item. A query on an item having rank r proceeds as follows: promote the accessed item to the top levels of the skip list, set its MTF-rank to 1, demote $\Theta(\log_2 r)$ items to lower levels (selected according to the current MTF-ranks), and increment the $\Theta(r)$ MTF-ranks of all the 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.

Consider generalizing the BSL for external memory operations. Say we adopt the bucketing strategy of [4] to make BSL work in external memory. BSL now consists of $O(\log_B n)$ levels and *seems* to take $\Theta(\log_B r)$ expected amortized I/Os per operation. However this is not true because we need to update BSL and MTF-list consistently after a query operation, and this is a challenge since 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 skip list, 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 those lists. The split/merge of one page needs the modification of $\Theta(B)$ pointers, namely the ones pointing to the items belonging to the affected page and coming from *arbitrary positions* (i.e. disk pages) of the MTF-list. Hence each split/merge operation on the skip list may require $\Theta(B)$ I/Os, thus resulting in a $\Theta(B \log_B r)$ I/O-bound which is suboptimal. Other natural strategies in this context also lead to suboptimal bounds. The preceding discussion applied to dictionary comprising numeric values. With string dictionaries, the challenges are only more severe.

In what follows, we introduce a novel variation of skip lists for solving our problem of external memory string dictionary. Some terminology follows. The *column* of an item is the "list" formed by all the elements in $\mathcal{L}$ which contain that item and are linked via the "vertical" pointers that connect adjacent levels of $\mathcal{L}$. The *height* $h(x)$ of an item x is the length of the column of x. Further, since we number the lists in $\mathcal{L}$ from top to bottom, we define a list $\mathcal{L}_i$ *higher* then a list $\mathcal{L}_{i+k}$ for any positive k (similarly, $\mathcal{L}_i$ is *lower* $\mathcal{L}_{i-k}$).

3 SASL in Main Memory

We need *random* subsets of items in successive levels of the skip list for balancedness. We also need to force the frequently accessed items to reside in the higher levels in order to make the skip list self-adjusting. We achieve these two apparently antithetic goals via the following approach: the column of each item in the skip list is forced to have two parts— one randomized and one deterministic. The randomized part is generated like in the original skip list (i.e., via coin tosses) and the deterministic part adapts itself continously to the sequence of searches/updates.

We define $f(x) = \sum_{i=0}^{x-1} 2^{2^i}$ as the maximum number of objects we can insert into x buckets of doubly-exponential size. Given an integer n, we then define $b(n)$ as the integer such that: $f(b(n)-1) < n \leq f(b(n))$. Clearly $b(n) = \Theta(\log\log n)$. The meaning and value of these functions will be revealed as the description of the data structure proceeds.

Our novel skip list $\mathcal{L}$ is formed by $H = \Theta(\log n)$ lists: $\mathcal{L}_1, \ldots, \mathcal{L}_H$; the lists satisfy the (classical) enclosure property $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$, and $\mathcal{L}_H$ is formed by all items in $\mathcal{S}$. However, the items are now distributed among these lists via a special strategy which combines *deterministic* and randomized coin tosses, and is further driven by the logical decomposition of the skip list described below.

The lists of $\mathcal{L}$ are logically decomposed into groups whose sizes increase exponentially. Each group consists of a set of adjacent lists in $\mathcal{L}$, hereafter called *(horizontal) bands*. We denote the bands with $\mathcal{B}_1, \mathcal{B}_2, \ldots$. The first band $\mathcal{B}_1$ consists of the list $\mathcal{L}_1$, the second band $\mathcal{B}_2$ consists of the next two lists $\mathcal{L}_2$ and $\mathcal{L}_3$; generally, the i-th band $\mathcal{B}_i$ consists of 2^{i-1} adjacent lists, namely $\mathcal{L}_{2^{i-1}} \cdots \mathcal{L}_{2^i-1}$ (the *width* of $\mathcal{B}_i$ is therefore 2^{i-1}). The number of these bands is actually $b(n)$. We have:

1. The *height of a band* $\mathcal{B}_i$, defined as the total number of lists belonging to lower bands (i.e. the overall number of lists in $\mathcal{B}_{i+1}, \ldots, \mathcal{B}_{b(n)}$), is equal to $H_i = 2^{b(n)} - 2^i$. Hence the height of the first (higher) band $\mathcal{B}_1$ plus its width (i.e. 1) gives the height $H = H_1 + 1 = \Theta(\log n)$ of the skip list $\mathcal{L}$.
2. In the band $\mathcal{B}_i$, the upper list is $\mathcal{L}_{2^{i-1}}$ and the lower list is $\mathcal{L}_{2^i-1}$. Hence we define the two parameters $u(i) = 2^{i-1}$ and $l(i) = 2^i - 1$.

For the sake of presentation in Appendix we show a figure to detail an example of our self-adjusting skip list. The logical decomposion of $\mathcal{L}$ in bands is used to drive the distribution of $\mathcal{S}$'s items among the various lists, always guaranteeing the condition that $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$. We associate two integer quantities with each item s: A *random* value $r(s)$ indicating the number of times head came out in a sequence of coin tosses on s; a *deterministic* value $d(s) = H_i$, for some i, thus forced to be equal to the height of some band. Hence $d(s)$ represents a sort of *deterministic promotion* of the item s to the band $\mathcal{B}_i$; after that, s possibly "proceeds" to higher lists via the classical random promotion (i.e. via the value $r(s)$).

As a result, the *height* of an item s in the skip list is now defined as $h(s) = r(s) + d(s)$, thus consisting of a deterministic part and a randomized part. This way, the item s is allocated in the lists $\mathcal{L}_{H-h(s)+1}, \mathcal{L}_{H-h(s)+2}, \ldots, \mathcal{L}_H$. For the sake of presentation we bound $h(s)$ to be at most H by setting: $h(s) = H$ in the case that $r(s) + d(s) > H$. The figure in the appendix shows an example.

Three observations are in order at this point: (1) the deterministic part of a column cannot end inside a band but always *synchronizes* to its beginning; (2) the integer $d(s)$ is not fixed once forever but it changes over the time as the search/update operations are executed on the data structure (it will help $\mathcal{L}$ to adapt itself to the observed frequencies of the accessed items); (3) the "truncation" of the long columns to H does not affect our mathematical analysis because it can be proven, by adopting standard techniques [15], that the number of truncated columns is a *constant* (so they could be managed specially).

According to the decomposition of $\mathcal{L}$ and the definition of the item heights we can introduce the following characterization of the distribution of items:

1. We say that item s *is born* at the band $\mathcal{B}_i$ if $d(s) = H_i$ (i.e. its deterministic part promotes s at the beginning of $\mathcal{B}_i$). We use variable B_i to denote the number of items that are born at band $\mathcal{B}_i$.
2. We say that item s *is alive* in the band $\mathcal{B}_i$ if it is born at a (strictly) 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 $\mathcal{B}_i$). We use variable A_i to denote the number of items which are alive at $\mathcal{B}_i$.
3. Given a band $\mathcal{B}_i$, we use variable W_i to denote the number of items which *will be born* in bands higher than $\mathcal{B}_i$, that is, they are born at $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$. (The deterministic part of the column of these items passes through band $\mathcal{B}_i$.)

Of course an item can live in more than one band if $r(s)$ is sufficiently large. Additionally, an item can belong *deterministically* to many bands if it is born in a high band. The characterization above therefore identifies all items whose column passes through a given band: either because of a long deterministic part of the column (captured by W_i) or because of a long randomized part (captured by A_i and B_i).

At a first glimpse it might seem that such a deterministic promotion strategy jeopardize the nature of skip lists. However, by further *constraining* the number of items which are born in a band, we can actually prove that: (1) every band *behaves* as it were a *single skip list*, thus inheriting all the nice properties of this data structure, and (2) the deterministic part of every item's column acts as an *anchor* between adjacent bands thus resulting useful to make incremental searches among them (*à la fractional cascading* [5, 12]). More precisely, we have.

The number of items which are born at a band $\mathcal{B}_i$ is forced to be $B_i = 2^{2^{i-1}}$ (i.e. it grows doubly exponentially in the band-number). In the lowest band $\mathcal{B}_{b(n)}$ we eventually have $B_{b(n)} \leq 2^{2^{b(n)-1}}$, so that $b(n)$ correctly identifies the number of bands in a SASL built over n items. Further since the width of band $\mathcal{B}_i$ is $2^{i-1} = \Theta(\log_2 B_i)$, we can now surprising infer that the random variable A_i has a constant expected value (this holds with high probability too). Hence the bands below $\mathcal{B}_i$ do not contribute significantly to the number of items alive at $\mathcal{B}_i$. So that, if we do not consider the W_i items which deterministically pass through $\mathcal{B}_i$, we can intuitively look at this band as an *independent skip list* (proof in the Appendix).

Theorem 1 *The expected number of items born and alive in the band $\mathcal{B}_i$ is $\mathbb{E}[A_i + B_i] \leq 2 + B_i$.*

We will show below that the deterministic columns in $\mathcal{B}_i$ created by those W_i items will play no role in the search operation, so that each band will be a real and independent skip list with all its nice structural properties. Furthermore, since $W_i = \sum_{k=1}^{i-1} B_k = \Theta(B_i)$, the extra space occupied by the deterministic columns is asymptotically no more than the space occupied by classical skip lists.

In order to implement our self-adjusting strategy, we need to slightly modify the classical search procedure on skip list [15] by introducing some *local restructuring* operations. They are obtained by moving items among the bands: promoting the most frequently accessed items to the higher bands (thus facilitating their future retrieval), and keeping the "population" of each band almost fixed along the sequence of queries (thus preserving its structural properties highlighted above). This means that each access to an item x, born at band $\mathcal{B}_j$, triggers its promotion to the highest band $\mathcal{B}_1$ and forces the consequent demotion of one item born at $\mathcal{B}_i$ to the immediately lower band $\mathcal{B}_{i+1}$, for each $i \in [1, j)$. The demotion of an item s from band $\mathcal{B}_i$ to band $\mathcal{B}_{i+1}$ actually consists of changing the "deterministic part" of its column from $d(s) = H_i$ to $d(s) = H_{i+1}$. The random value $r(s)$ remains unaltered during all the operations.

Changing the values $d()$ for a subset of $\Theta(j) = O(\log \log n)$ items has an impact on the structure of $\mathcal{L}$ because we have to change the columns relative to these items of the corresponding amount. Namely, we need to insert the item s in all the lists $\mathcal{L}_1, \ldots, \mathcal{L}_{l(j)}$ and remove each item selected to be demoted from $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$, from exactly $H_i - H_{i+1} = 2^i$ lists (where $i \in [1, j)$) . Once the items to be demoted have been found, their removal takes just $\Theta(1)$ time per affected list because of the presence of the "vertical" pointers between lists in adjacent levels of $\mathcal{L}$. Hence, the overall cost is proportional to the number of elements removed/inserted from/to the lists of the bands above $\mathcal{B}_j$ (which contains x). This cost is expressed by the following formula: $O(\sum_{i=1}^{j-1}(H_i - H_{i+1})) = O(H_1 - H_j) = O(2^j)$ and thus is proportional to the width of the band $\mathcal{B}_j$ containing the accessed item.

In the rest of the section we complete the description of the query operation by addressing three issues: (1) how to determine the items to be demoted, (2) how to search for the accessed item x, and finally (3) what is the expected position of an item as the randomized demotion procedure is applied during the life-time of $\mathcal{L}$.

Random select. As we briefly mentioned in the introduction of this paper our demotion procedure is based on a random selection which exploits *locality of reference* to minimize the number of page faults elicited in this process. The main idea is to design an algorithm which is able to select *uniformly at random* one of the items born at a given band $\mathcal{B}_i$. The difficulty in performing this task resides in the fact that we do not want to keep extra-pointers to identify the items actually born in that band because their management might be costly in a external memory setting (see Section 2).

Hence we introduce some local information that drives the random retrieval of such an item from $\mathcal{B}_i$. Let s be an item of $\mathcal{L}$, and let $\mathcal{B}_j$ be the band where s is born. At the top of s's column we store a set of integers $c_{j+k}(s)$ such that $c_{j+k}(s)$ counts the total number of items which are born at a lower band $\mathcal{B}_{j+k}$ and whose columns are stictly lower than the one of s. Intuitively $c_{j+k}(s)$ counts the items that are born in $\mathcal{B}_{j+k}$ and are reachable from s via the classical skip-list search procedure. Item s therefore stores $O(\log\log n)$ integers, each requiring $O(\log_2 c_{j+k}(s)) = O(\log_2 B_{j+k}) = O(2^{j+k})$ bits; summing up for all positive k we derive that $O(2^{b(n)}) = O(\log n)$ bits per item are sufficient. This amount of bits is bounded above by usual the space needed to store the skip-list pointers, i.e. $O(n \log n)$ bits.

To select, uniformly at random, the item to be demoted from the band $\mathcal{B}_i$ we use the c_is. These values drive a *random walk* in $\mathcal{L}$ that ends up at an element on the lowest list $\mathcal{L}_{l(i)}$ of the band $\mathcal{B}_i$. More precisely, we perform a search-like algorithm on $\mathcal{L}$ such that, at an item s, it chooses to go rightward (resp. downward) with a probability that *depends* on the number of items that are born at $\mathcal{B}_i$ and are reachable by following the forward (resp. downward) pointer at s. The number of items that are reachable by following the current downward pointer are stored in $c_i(s)$; the ones reachable by the current forward pointer may be computed incrementally as the random walk proceeds by exploiting the knowledge of $c_i(s)$ and B_i.

Theorem 2 *The algorithm* **Random_Select**$(\mathcal{B}_i)$ *returns an item picked up uniformly at random among the ones born at $\mathcal{B}_i$. The expected time is $O(2^i)$ and the extra space needed to store the auxiliary information is $O(n \log n)$ bits.*

Search for an item x in $\mathcal{L}$. This is exactly as the search-procedure for classical skip-lists [15]. The correctness is therefore immediate, whereas the analysis of the time complexity needs some considerations. We already observed that our data structure can be seen from an high-level point of view as a *cascade* of skip lists (one per band), having a doubly-exponentially growing number of items. These skip lists are melded together via the use of the *deterministic* part of the items' columns. More precisely, the randomized part of the column of an item plays the standard role of items' columns in skip lists; conversely, the deterministic part of that column plays the role of an *anchor point* in the lower skip lists (i.e. lower bands). As a consequence, the time spent in a band to search for an item is not lost if that item in not found in that band, but the last visited item can be used as a *finger* into the next skip list to continue the search operation. The reader familiar with the *fractional cascading* technique [5, 12] should find many similarities here.

Since the search algorithm never crosses a deterministic part of a column, the search in $\mathcal{L}$ traverses only the random parts so that we can apply standard techniques [15] to prove the time bounds for our data structure. This means that searching for an item x in a band $\mathcal{B}_j$ takes $O(\log_2(A_j + B_j)) = O(2^j)$ expected time (see Theorem 1). Since the search proceeds from the topmost band $\mathcal{B}_1$ to the band actually containing x, say $\mathcal{B}_i$, the overall cost is $O(\sum_{j=1}^{i} 2^j) = O(2^i)$ expected time.

Theorem 3 *The search for x in $\mathcal{L}$ and its consequent restructuring takes $O(2^i)$ expected time, if $\mathcal{B}_i$ is the band where x is born. The space occupied by the SASL is $O(n \log n)$ expected bits.*

We point out that the restructuring of $\mathcal{L}$ requires the change of some c_js due to the promotion/demotion operations subsequent to the search for x. The changed values actually lie on the search path for x and thus their modification can be performed within the same time bound.

Performing a sequence of searches. Since the "native" band of an item changes as searches are executed on the data structure as the result of the restructuring operations (i.e. random demotions and promotions), we conclude that, in order to estimate the overall cost of a sequence of searches, we need to compute in expectation the band which contains the accessed item. Although a frequently accessed item may be unfortunately demoted to lower bands, because of **Random_Select**, the probability that this happens decreases exponentially as we move from the higher to the lower bands because of their doubly-exponential increase in size. Therefore, on the average, the position of an item will be the one that it would occupy if the demotion procedure would be driven by an LRU-strategy (Least Recently Used). The proof of the following theorem is given in the Appendix.

Theorem 4 *In a SASL built over n items, a sequence 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.*

The insertion of an item s is always performed in the topmost band by setting $d(s) = H_1$ and by flipping a coin to compute $r(s)$. Since the number of items born into a band is fixed, the insertion triggers a cascade of demotions which eventually end up into the bottommost band thus taking overall $O(\log n)$ expected time.

The deletion algorithm consists of removing the column of the deleted item s, and updating consistently all the affected lists and values c_js (this can be done simultaneously to the search for s). If s was born in, say, $\mathcal{B}_k$ we also need to perform a promotion from band $\mathcal{B}_{k+1}$ by taking a random item from it. The cost of a deletion is then $O(\log n)$ expected time.

4 SASL in External Memory for Strings

Skip lists can be easily adapted to work in external memory by setting the probability of flipping a head as $\Theta(1/B)$ [4]. This way, the height of the skip list is $O(\log_B n)$. It is not difficult to generalize the algorithms we designed for our SASL from the previous section to work in this context too. Just observe that any restructuring of our data structure is done during a downward traversal so that pointers and disk pages can be updated efficiently by exploiting locality of reference.

The external-memory SASL can be further adapted to manage *string items* as well. We do not copy the item every time it occurs in $\mathcal{L}$ but we just use pointers to its copy residing on disk. This clearly introduces one level of indirection, thus forcing us to access the disk anytime two strings (i.e. two pointers to strings) have to be compared. We show that we can amortized the cost of string comparison by accessing the disk only when this is absolutely needed (this is quite similar to string B-trees [7]). We briefly summarize the issues here.

Let us consider an element s in a skip-list level $\mathcal{L}_i$ pointing to a string α, and let $pred(\alpha)$ (resp. $succ(\alpha)$) be the strings stored in the predecessor (resp. successor) of s in $\mathcal{L}_i$. We augment our data structure in two ways: (1) we store the longest common prefix length $lcp(\alpha, succ(\alpha))$ together with the pointer to string α, (2) make the lists $\mathcal{L}_i$ doubly linked via pointers *succ* and *pred*.

Let P be the string to be searched and suppose we have function $extend\_lcp(l, P, \alpha)$ that computes the *lcp* between the two strings P and α knowing that they have at least l characters in common, i.e., the function begins the comparison of the two strings from their $(l+1)$-th character. Searching for string P in the external-memory SASL $\mathcal{L}$ proceeds per levels by maintaining the following invariant: L_i and R_i are the two strings that delimit 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 M_i is the string between L_i and R_i which shares the longest prefix with P. The peculiarity of the search at level $\mathcal{L}_i$ is that it can go forward or backward over the items in $[L_i, R_i]$ depending on the *lcp*-value among M_i and P. If we need to pay I/Os in order to compare P with the strings in $[L_i, R_i]$, these I/Os gather new information on the longest prefix shared by P, and thus can be saved in the next comparisons by the use of *extend_lcp* procedure. Essentially as in [7], we can show that the additional I/O-cost incurred by the search procedure in string scanning is bounded above by $O(1 + |P|/B)$ I/Os (details in the Appendix). Hence the first main result of our paper follows:

Theorem 5 *In an external-memory self-adjusting skip list 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 of length $|S_i|$ is queried. Inserting or deleting a string S takes $O(\frac{|S|}{B} + \log_B n)$ expected I/Os. The total space occupied by our data structure is $O(N/B)$ expected disk pages.*

5 Exploiting Main Memory as a Cache

The I/O-bound provided in Theorem 5 exploits actually one aspect of the self-adjusting issue: the frequency of string accesses in the query sequence $\mathcal{Q}$. The *string nature* of the managed items is not exploited and, in fact, strings are self-adjusted as if they were *atomic items* by just moving their pointers around. However if the query sequence $\mathcal{Q}$ has search for two strings *String-algorithmic* and *String-library*, we would like our algorithm to take advantage of the prefix *String-* (that is shared between them) when searching for the second string. Usually this repetitiveness is exploited by the *buffering/caching* policies of the underlying operating system; here, we wish to extend our data structure and its analysis in order to explicitly take this into account. In particular, we propose that the main memory be used persistently across queries so that it behaves as a cache, and we design caching policies integrated into SASL. Sketch of the details follow.

We assume we have a *cache* $\mathcal{C}$ of size c disk pages in the main memory. The cache contains 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 $j \leq i$, we keep a pointer to the *topmost* string in the self-adjusting skip list having that prefix. "Topmost" here means that the pointed string is the first one we would encounter when searching in the skip list for a string prefixed by $S[1, jB]$. These pointers represent a sort of *"shortcuts"* into the SASL to speed up the search operations when a prefix of the queried string is available in the cache $\mathcal{C}$.

Let S be the currently queried string. We search in $\mathcal{C}$ for the longest cached prefix of S, let it be $S[1, iB]$. We follow the pointer associated to this prefix and thus reach the topmost string in the SASL prefixed by

$S[1, iB]$. We then continue the search in the SASL for the remaining suffix $S[iB+1, |S|]$ as described in Section 4. This means that we have saved i I/Os in the search process (specifically the ones due to the scanning of the first i pages of S). Our aim will be then to estimate the average value of i as a function of the "regularity" (i.e. entropy) of the query sequence and the size of $\mathcal{C}$.

Before proceeding with these calculations we detail the policy we use to manage $\mathcal{C}$ after S has been queried. We extend the cached prefix $S[1, iB]$ by loading in cache the next page of S, i.e. $S[iB+1, (i+1)B]$. Successively we set properly the pointer associated to this longer prefix, noting that the pointed string must lie on the skip list path followed in the search for $S[iB+1, |S|]$. To make room in $\mathcal{C}$ for the loaded page we need to evict some other page. The evicted page is chosen among the oldest final pages of the cached prefixes; this way some cached prefix has one fewer page.

It remains to be shown how these shortcut pointers are kept under promotion/demotion operations, as well as insertion/deletion of strings in the SASL. We defer the full discussion to the final version of this paper and just sketch the demotion case here, the other cases being similar. The demoted string might be the destination of some shortcut pointers that, therefore, may have to be moved. They may only move from the demoted string to its adjacent one to the right in the same level of the skip list. Testing if this is the case can be done by exploiting the knowledge about the longest-common prefix information kept between adjacent strings on the same level. Hence the movement of each shortcut pointer can then be executed together with the demotion operation without requiring any further I/Os.

The following theorem (proof in the Appendix) measures the goodness of our caching strategy as a function of the entropy of string prefixes in the query sequence $\mathcal{Q}$, and the cache size c. We actually introduce a parameter $\gamma(S_{i_j})$ that accounts for the average number of cache misses, and recall the definition of entropy $\mathcal{H}$ of a set of x items having probability $p_1, \ldots, p_x$, i.e. $-\sum_i p_i \log_2 p_i$.

We point out that in the next theorem we are assuming strings of length smaller than the cache size; we will comment on the general case below.

Theorem 6 *Let $S_1, \ldots, S_n$ be a set of n strings having total length N, and $\mathcal{Q}$ be a query sequence of length m: $S_{i_1}, S_{i_2}, \ldots, S_{i_m}$. The combination of the self-adjusting skip list with the cache $\mathcal{C}$, of size c disk pages, allows us to answer $\mathcal{Q}$ in*

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

amortized, expected I/Os over all m searches, 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 the 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 this data structure is $O(N/B)$ disk pages.

Notice that if the query sequence $\mathcal{Q}$ is repetitive, then the entropy $\mathcal{H}[r] \approx 0$ for all r. Hence the theorem allows us to conclude that in such cases the whole query string S_{i_j} is cached, and thus no I/Os are incurred by the scanning of the string S_{i_j}. I/Os for SASL rebalancing are still needed and accounted in the second term above (see Concluding Remarks for further discussion).

In the case of a very long string S_{i_j}, we pay I/Os for its pages in the suffix $S_{i_j}[\alpha|\mathcal{C}|+1, |S_{i_j}|]$, where α is a positive constant smaller than one. For the cost of accessing the prefix $S_{i_j}[1, \alpha\,|\mathcal{C}|]$ we can still use the bound in Theorem 6.

6 Concluding Remarks

We have initiated the study of data structures for external memory string access. We have designed a novel self-adjusting skip lists (SASL) that adapts itself to the distribution of string queries without explicitly setting weight or structural constraints. We proved the Static Optimality Theorem for external memory string accesses, and thus showed that our SASL is optimal with respect to the best offline static string index. We further improved on the performance of this data structure by introducing the paradigm of using (part of) the main memory persistently across the queries essentially as a cache to decrease disk page accesses dependent on the entropy of string prefixes in the queries. This changes the "nature" of the Static Optimality Theorem in the presence of memory hierarchies and string data.

Many interesting questions remain. For one, we leave open the problem of proving the Static Optimality Theorem deterministically (our SASL data structure uses randomization to construct the skip list as well in self-adjusting by demotion of items). Our work can be extended to multiple levels of the memory hierarchy in a straightforward way. There has been recent focus on designing "cache-oblivious" algorithms, that is, ones which work without knowledge of the parameters of the memory hierarchy (such as B). Extending the cache-oblivious B-trees of [2], we can obtain cache-oblivious algorithms for the string dictionary problem we have studied here. The details are not straightforward and leave its discussion for later. Finally, there are other forms of self-adjusting results that will be of interest: working set theorem [17], dynamic optimality [3, 17], and reference locality [6].

SASL allows us to also answer *range queries* on pairs of strings (S, S'): retrieve all the dictionary words which lexicographically lie between S and S'. The algorithm is simple due to the structure of a skip list. The overall performance and impact of SASL in practice remains to be studied.

We would also like to apply our paradigm of using the main memory as a cache for a sequence of operations. For example, a goal would be to extend our result to *substring* accesses (answer queries of the form, is there a substring of the dictionary strings equals to the query pattern?). From an algorithmic point of view, this is not difficult since we can index all the suffixes of the strings (*à la* String B-tree), but the caching strategy is unclear.

Some open problems remain concerning the use of main memory as a cache. In [19] optimal prefetching algorithms have been presented and analyzed, deriving them from compression algorithms. The analysis or algorithms there might probably be extended to achieve tighter I/O-bounds.

An extension of our work here is to cache the top $\Theta(\log_B M)$ levels of the external memory SASL, where M is the main memory size. This caching of the data structure (in addition to the string prefixes we cached in Section 5) will further help in query sequences that are highly repetitive. Details will be provided in the final version of this paper.

References

[1] N. Abramson. *Information Theory and Coding.* McGraw Hill, 1983.

[2] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *41st Annual Symposium on Foundations of Computer Science*, pages 399–409. IEEE Computer Society Press, 2000.

[3] A. Blum, S. Chawla, and A. Kalai. Static optimality and dynamic search-optimality in lists and trees. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002 (to appear).

[4] P. B. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Algorithms and Data Structures, 4th International Workshop*, volume 955 of *Lecture Notes in Computer Science*, pages 381–392, 1995.

[5] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[6] F. Ergu, S. C. Sahinalp, J. Sharp, and R. K. Sinha. Biased dictionaries with fast insert/deletes. In *ACM Symposium on Theory of Computing*, pages 483–491, 2001.

[7] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, Mar. 1999.

[8] R. Grossi and G. Italiano. Efficient Techniques for Maintaining Multidimensional Keys in Linked Data Structures. In *Procs International Colloquium on Automata, Languages and Programming*, pages 372–381, 1999.

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

[10] C. Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, May 1991.

[11] K. Mehlhorn. *Data Structures and Algorithms 1: Searching and Sorting*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1984.

[12] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[13] K. Mehlhorn and S. Naher. Algorithm design and software libraries: Recent developments in the leda project. In *IFIP Congress*, volume 1, pages 493–505, 1992.

[14] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[15] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[16] M. Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, July 1995.

[17] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[18] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 2001 (To appear).

[19] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, 1996.

A A Useful Picture

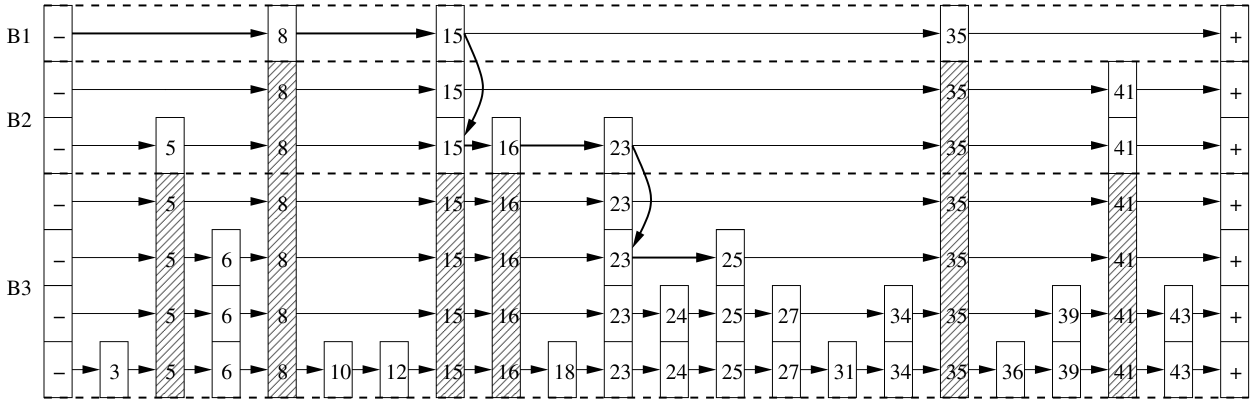

Figure 1: An example of SASL and a search for the item 25 into it. The shadowed (lowest) part of a column is the deterministic one. Here $b(n) = 3$ so that $\mathcal{L}$ is decomposed into three horizontal bands $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ with $H = 7$. Notice that $u(1) = 1, u(2) = 2$ and $u(3) = 4$; in fact $\mathcal{L}_1, \mathcal{L}_2$ and $\mathcal{L}_4$ are the highest lists in each band. Item 16 has $r(16) = 1$ and $d(16) = 4$, hence $h(16) = 5$.

B Some Proofs

Theorem 1. *The expected number of items born and alive in the band $\mathcal{B}_j$ is $\mathbb{E}[A_j + B_j] \leq 2 + B_j$.*

Proof. Let us define the event $A_{j,j+k}$ denoting the number of items that are born at the band $\mathcal{B}_{j+k}$ (with $k > 0$) and are alive at the band $\mathcal{B}_j$. Each one of these items, say item s, must have a random part $r(s)$ long enough to span the bands $\mathcal{B}_{j+k}, \mathcal{B}_{j+k-1}, \ldots, \mathcal{B}_{j+1}$. This actually means that it must be $r(s) > H_j - H_{j+k} = 2^{j+k} - 2^j = 2^j(2^k - 1)$. The probability of this event is $\mathbb{P}(r(s) > H_j - H_{j+k}) = (1/4)^{2^j(2^k-1)}$. Summing up on all bands which lie below $\mathcal{B}_j$, and recalling that the number of items born at $\mathcal{B}_{j+k}$ is $B_{j+k} \leq 2^{2^{j+k-1}}$, we get by simple algebraic calculations:

$$\mathbb{E}[A_j] = \mathbb{E}[\sum_{k=1}^{b(n)-j} A_{j,j+k}] = \sum_{k=1}^{b(n)-j} \mathbb{E}[A_{j,j+k}] \leq \sum_{k=1}^{b(n)-j} B_{j+k} \times (1/2)^{2^j(2^k-1)} \leq \sum_{k=1}^{b(n)-j} (1/2)^{2^j(2^{k-1}-1)} \leq 2$$

■

Theorem 3. *The search for an item x in $\mathcal{L}$ and the consequent restructuring takes $O(2^i)$ expected time, if $\mathcal{B}_i$ is the band where x is born. The total space occupied by the SASL is $O(n \log n)$ expected bits.*

Proof. We know that the retrieval of x and the identification of its native band takes $O(2^i)$ expected time. After that x is promoted to the first band $\mathcal{B}_1$, by extending its column of $H_1 - H_i = O(2^i)$ elements, thus taking $O(2^i)$ time complexity. Subsequently, algorithm **Random_Select**$(\mathcal{B}_j)$ is executed on each $j = 1, 2, \ldots, i-1$ to keep constant the number of items born at each one of these bands. From Theorem 2, algorithm **Random_Select**$(\mathcal{B}_j)$ takes $O(2^j)$ expected time. From the observations in Section 3 we know that the demotion of the item selected in $\mathcal{B}_j$ takes $O(2^j)$ expected time, and thus overall $\sum_{j=1}^{i-1} O(2^j) = O(2^i)$ expected time is needed to demote all those items. We finally need to update the values of c_js as the result of the restructuring operations, some elements changed native-band. It is easy to see that the number of variables changing their c_js value is upper bounded by the number of elements traversed during the execution of **Random_Select**. Additionally, the change induced on some $c_j(s)$ is indeed known at the time the search for the demoted item passes through item s, thus it can be executed istantaneously. Since the number of items traversed by **Random_Select**$(\mathcal{B}_j)$ over all $j < i$ is $O(2^i)$ (see above), we can conclude that this is actually the cost of $\mathcal{L}$'s restructuring.

As far as the space occupancy is concerned, by the observations in Section 3 the auxiliary space needed to store the c_js values is $O(n \log n)$ bits. The space needed by the random part of the items' columns is $O(n \log n)$ expected bits [15]. The space occupied by the deterministic part of the items' columns is bounded above by $O(\sum_{j=1}^{b(n)} B_j \times H_j \times \log n) = O(n \log n)$ bits. ■

Theorem 4. *In a SASL built over n items, a sequence 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. Here we slightly follow the proof of Theorem 3.3 in [10], with the further difficulty that the number of the items alive in a band is not fixed but it is a random variable (with constant expectation, see Theorem 1). Let us concentrate on an item s, and consider the first istant, say T, item s is accessed. (In generral, we denote by T the number of operations executed since the last look up to s.) Of course T is an upper bound to the number of *distinct* items accessed in the sequence until that instant. Due to our deterministic promotion to $\mathcal{B}_1$ of the accessed items, and because of the doubly-exponential increasing size of the bands (i.e. $B_i = 2^{2^{i-1}}$), we can immediately conclude that item s at instant T must be born at a band $\mathcal{B}_i$ where $i \geq \log\log T$. If we would always have $i = \log\log T$ (i.e. s is chosen once per band to be randomly demoted), our self-adjusting strategy would behave exactly as the MTF-startegy [6]. But here we may have some variance from the "best" band, which in any case can be proven to be bounded to a constant, in an expected sense.

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

Consequently, the probability that an item s after T accesses is born at a band $\mathcal{B}_{k+j}$, with $k = \log\log T$, is surely less than the probability that in this T accesses s was in $\mathcal{B}_{k+j}$ and has been never demoted from there:

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

From Theorem 3 and since $T = 2^{2^k}$, we derive the expected cost for looking up item s after T accesses to other items:

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

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

Let us now consider the overall sequence of lookups to item s, and denote by $T_1, \ldots, T_{n_s}$ the number of operations between two of these lookups. Because of the concavity of the logarithmic function, the summation $\sum_{j=1}^{n_s} O(\log T_j)$ is maximize when all T_js are equal, that is $T_r = m/n_s$. Therefore the expected cost to access n_s times the item s in a sequence of m operations is $O(n_s \log \frac{m}{n_s})$. Summing up on all items we get the bound stated in the theorem. ■

C SASL for strings

The structure of a skip list is suitable to be adapted to manage also *string items*. Let α and β be two strings of arbitrary length, with $lcp(\alpha, \beta)$ we will denote hereafter the length of their longest common prefix. In [8] the notion of *lcp* has been used to adapt poiter-based data structures to the management of strings. In the present paper we follow a simpler approach driven by the structure properties of skip lists.

Since an item is now a string of arbitrary length, we do not copy the item every time it occurs into a list of $\mathcal{L}$ but we just use pointers to its copy on disk. This clearly introduces one level of indirection, thus imposing us to access the disk anytime two strings (i.e. two pointers to strings) have to be copared. Nonetheless we will show that we can amortized the cost of string comparison thus accessing the disk only when this is really needed (similar to string B-trees [7]).

Let us therefore consider an element s in the list $\mathcal{L}_i$ pointing to a string α_i, and let $pred(\alpha_i)$ (resp. $succ(\alpha_i)$) be the strings stored in the predecessor (resp. successor) of s in $\mathcal{L}_i$. We augment our data structure in two ways: (1) we store the value $lcp(\alpha_i, succ(\alpha_i))$ together with the pointer to string α_i, (2) make the lists $\mathcal{L}_i$ doubly linked (hence pointers $succ()$ and $pred()$ do exist).

Let P be the string to be searched in a collection of n string. Suppose we have 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 characters in common, i.e. the function starts to compare the two strings from their $(l+1)$-th chararacter. We now show how to perform the basic step of the search operation in a skip list, after that what remains to do is just to apply repeatedly it until the string P is found.

Our inductive assumption defines L_i and R_i as the two strings that delimit the portion of list $\mathcal{L}_i$ where the search has been confined by the previous search-steps on $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$. Furthermore, we set M_i as the string between L_i and R_i which shares the longest prefix with P, and store $\texttt{len}_i = lcp(M_i, P)$. The peculiarity of the string search procedure is that it can go forward or backward over the items in $[L_i, R_i]$ depending on some conditions on L_i, R_i, M_i.

As usual in skip lists, we check whether $succ(L_i) = R_i$, 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 $M_{i+1} = M_i$.

Let us assume now that $succ(L_i) \neq R_i$. Unlike classical skip lists, the next item to compare against P is chosen between $succ(L_i)$ and $pred(R_i)$ depending on the current value of M_i. Indeed we know $l_i = lcp(P, M_i)$ by the inductive search and thus take $x = succ(L_i)$ if $M_i = L_i$, otherwise $x = pred(R_i)$. In the data structure is available the value $l' = lcp(M_i, x)$ and we can get in $O(1)$ I/Os. Essentially as in [9, 7], 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 $L_i = M_i$.

As in [9, 7], we can show that at most one character of P is compared more than once during each step of the search procedure. If $q = extend\_lcp(l_i, P, x)$, it is easily seen that the I/O-cost of this computation is proportional to $1 + (q - l_i)/B$, which gives along the whole search operation a telescopic sum bounded above by $O(1 + |P|/B)$ I/Os. All the other computations involve local informations and thus can be performed within the same I/O-bounds of SASL in external memory. Of course, the updating of the data structure as a consequence of an insertion, deletion, or restructuring operation, results now more complicated. We leave the details to the reader and just observe here that the updating of the $lcp()$ information can be performed during the downward traversal of the list similarly contestually with the update of the bi-directional pointers, by exploiting the alphabeting ordering of the strings in $\mathcal{L}$.

D Proof of Theorem 6

We do not evaluate directly the behavior of our caching strategy but consider another approach, which is more complicated to be implemented but simpler to be analyzed. The new caching approach consists of loading in $\mathcal{C}$ the page $S[iB+1, (i+1)B]$ only if the prefix $S[1, (i+1)B]$ has been queried at least $(i+1)$ times during the last c string queries (recall that $|\mathcal{C}| = cB$). The probability of a cache hit in the new approach is smaller than the one in our caching strategy. Indeed any cache hit in the new approach guarantees that $S[1, iB]$ occurs at least i times in the last c string queries, so that our approach loads the jth page of S when querying the jth occurrence of $S[1, iB]$. Overall $S[1, iB]$ is in cache when S is queried.

Consequently by evaluating the probability of a cache miss in the new (stricter) caching policy we obtain an upper bound to the probability of a cache miss in our simpler caching approach.

Let us therefore concentrate on the stricter caching strategy. For the sake of presentation we introduce some useful quantities, where we assume to set $A = \sqrt{i/c} > (i/c)$:

- $p_i = \sum_{\substack{\text{all string prefixes } \sigma \\ |\sigma| = iB,\ p(\sigma) < A}} p(\sigma)$.
- $\mathcal{H}[i]$ = Entropy of the prefixes of length iB over the query sequence.

We now relate p_i and $\mathcal{H}[i]$:

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

¿From which we derive a bound on p_i as a function of the entropy $\mathcal{H}[i]$:

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

We wish to estimate the probability that the ith page of a query string S is not in cache. This means that the prefix $S[1, iB]$ of that string occurred at most $(i-1)$ times as a prefix of the previous c queried strings (recall that we are considering the stricter caching approach). To account for this number of accesses and thus estimate the above probability, 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. We can compute X_σ^c as a sum of Bernoulli variables defined as follows:

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

We can define $X_\sigma^c = \sum_{j=1}^{c} X_\sigma^c[j]$, and then 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.

$$\begin{aligned}
\mathbb{P}(\text{cache miss on the } i\text{th page}) &= \sum_{\sigma:\ |\sigma|=iB} \mathbb{P}(\text{cache miss on } i\text{th page}|\text{query prefix is } \sigma)\ \mathbb{P}(\sigma) && (1)\\
&= \sum_{\sigma:\ |\sigma|=iB} \mathbb{P}(X_\sigma^c < i) \cdot \mathbb{P}(\sigma) && (2)\\
&= \sum_{\substack{\sigma:\ |\sigma|=iB \\ \mathbb{P}(\sigma)<A}} \mathbb{P}(X_\sigma^c < i)\ \mathbb{P}(\sigma) + \sum_{\substack{\sigma:\ |\sigma|=iB \\ \mathbb{P}(\sigma)\geq A}} \mathbb{P}(X_\sigma^c < i)\ \mathbb{P}(\sigma) && (3)
\end{aligned}$$

The first term can be bounded as

$$\sum_{\substack{\sigma:\ |\sigma|=iB \\ \mathbb{P}(\sigma)<A}} \mathbb{P}(\sigma) = p_i \leq \frac{2\mathcal{H}[i]}{\log(c/i)} \qquad (4)$$

We now concentrate on finding an upper bound to the second term. Recall now the Chernoff bound for a sum of independent Bernoulli variables X_i, having average $\mathbb{E}[\sum X_i] = \mu$, it is $\mathbb{P}(\sum X_i < (1-\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}$. We use the Chernoff inequality 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:

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

We now assume that $i < \alpha c$, with $\alpha < 1$. Indeed we are concentrating on the caching of prefixes whose length is at most a constant fraction of the cache size. Given that, we can upper bound the previous formula as follows:

$$e^{-\frac{\sqrt{i\,c}}{2}(1-\sqrt{\frac{i}{c}})^2} \leq e^{-\frac{\sqrt{i\,c}}{2}(1-\sqrt{\alpha})^2} \leq e^{-\frac{\sqrt{i\,c}}{2}\beta} \tag{5}$$

where $\beta = (1-\sqrt{\alpha})^2$. Substituting the two upper bounds (4)–(5) in equation 1 we have:

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

According to our discussion above, we can calculate an upper bound to the expected number of cache misses incurred by our caching strategy (we set $r = 1$ in the first term):

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

Notice that for large c the first term is strictly smaller than 1 and thus we drop it. The formula captures the repetitiveness at prefix lengths smaller than $\alpha c B$.

Since we assumed that all the strings are shorter than the cache size, we can introduce the parameter $\gamma(S_{i_j}) = \min\{\frac{|S_{i_j}|}{B}, \sum_{r=1}^{|S_{i_j}|/B} \frac{\mathcal{H}[r]}{\log(c/r)}\}$ to capture the number of cache misses incurred by the search for S_{i_j}.