2006

Dynamic Rank/Select Dictionaries with Applications to XML Indexing

Ankur Gupta

Wing-Kai Hon

Rahul Shah

Jeffrey S. Vitter
Kansas University, jsv@ku.edu

Report Number:
06-014

DYNAMIC RANK/SELECT DICTIONARIES WITH APPLICATIONS TO XML INDEXING

Ankur Gupta
Wing-Kai Hon
Rahul Shah
Jeffrey S. Vitter

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #06-014
July 11, 2006

Dynamic Rank/Select Dictionaries with Applications to XML Indexing*

Ankur Gupta Wing-Kai Hon Rahul Shah Jeffrey Scott Vitter

Abstract

We consider a central problem in text indexing: Given a text T over an alphabet Σ, construct a compressed data structure answering the queries $char(i)$, $rank_s(i)$, and $select_s(i)$ for a symbol $s \in \Sigma$. Many data structures consider these queries for static text T [GGV03, FM01, SG06, GMR06]. We consider the dynamic version of the problem, where we are allowed to insert and delete symbols at arbitrary positions of T. This problem is a key challenge in compressed text indexing and has direct application to dynamic XML indexing structures that answer *subpath* queries [FLMM05].

We build on the results of [RRR02, GMR06] and give the best known query bounds for the dynamic version of this problem, supporting arbitrary insertions and deletions of symbols in T. Specifically, with an amortized update time of $O((1/\epsilon)n^{\epsilon})$, we suggest how to support $rank_s(i)$, $select_s(i)$, and $char(i)$ queries in $O((1/\epsilon)\log\log n)$ time, for any $\epsilon < 1$. The best previous query times for this problem were $O(\log n \log|\Sigma|)$, given by [MN06]. Our bounds are competitive with state-of-the-art static structures [GMR06]. Some applicable lower bounds for the partial sums problem [PD06] show that our update/query tradeoff is also nearly optimal. In addition, our space bound is competitive with the corresponding static structures. For the special case of bitvectors (i.e., $|\Sigma| = 2$), we also show the best tradeoffs for query/update time, improving upon the results of [MN06, HSS03, RRR02].

Finally, our focus on fast query/slower update is well-suited for a query-intensive XML indexing environment. Using the XBW transform [FLMM05], we also present a dynamic data structure that succinctly maintains an ordered labeled tree T and supports a powerful set of queries on T.

*Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066, USA ({agupta, wkhon, rahul}@cs.purdue.edu, jsv@purdue.edu).

1 Introduction

The new trend in text indexing data structures is to compress and index data in one shot. The ultimate goal of these compressed text indexes is to retain near-optimal query times (as if not compressed), yet still take near-optimal space (as if not an index). A few of the pioneer results in this area are [GV00, FM00, FM01, GGV03]; there are many others.

Progress in compressed text indexing has gone hand-in-hand with exploring similar goals for more combinatorial structures (like trees and subsets). For these *succinct* data structures, the emphasis is to store them in terms of the information-theoretic (combinatorial) minimum required space. Again, these structures aim to retain fast query time [RRR02, Jac89, BM99, Pag99, HMP00]. Compressed text indexing makes heavy use of succinct data structures for set data, or *dictionaries*. A recent result by [FLMM05] combines a succinct data structure for trees with a Burrows-Wheeler text compression scheme and achieves a compressed data structure for querying ordered labeled trees. This result has direct applications in XML compression and indexing [FLMM06].

To date, most compressed text indexing work is largely concerned with static data. However, many environments actually need compressed indexing functionality on dynamic data: XML documents and web pages, CVS projects, electronic document archives, etc. In these settings, we require a compressed dynamic index that is able to answer queries efficiently and still perform updates in a reasonable amout of time.

In that vein, there have been some results on dynamic succinct bitvectors (dictionaries) [RRR01, HSS03, MN06]. However, these data structures either perform queries in far from optimal time (in query-intensive environments), or allow only a limited range of dynamic operations ("flip" operations only). In this paper, we develop a compressed dynamic data structure called *BitIndel* that supports fast queries and *and* arbitrary insertion and deletion of bits. Our update/query tradeoffs are nearly optimal for indexing bitvectors. We define the *dynamic bit dictionary* problem: Given a dynamic bitvector B of length n, we allow the following operations for a bit $s \in \{0, 1\}$:

- $rank_s(i)$ tells the number of s bits up to the ith bit in B;
- $select_s(i)$ gives the position in B of the ith s bit;
- $insert_s(i)$ inserts s before the position i in B;
- $delete(i)$ deletes the ith bit from B.

For the static case, [RRR02] solves the bit dictionary problem using $nH_0 + o(n)$ bits of space and answers rank and select queries in $O(1)$ time, where H_0 is the 0th order empirical entropy of the bitvector B (considered as a string). The best known time bounds for the dynamic problem are given by [MN06], achieving $O(\log n)$ for all operations. In this paper, we introduce a new dynamic bit dictionary (called BitIndel) that takes $O((1/\epsilon) \log \log n)$ time for queries, while supporting updates in $O((1/\epsilon)n^\epsilon)$ amortized time.

Our main problem is a generalization of the bit dictionary problem to a larger alphabet Σ called the *text dictionary* problem. The queries supported are $rank_s$, $select_s$, and $char$, where s is any symbol in Σ. The text dictionary problem is a key tool in text indexing data structures. For the static case, Grossi et al. [GGV03] present a wavelet tree structure that answers queries in $O(\log |\Sigma|)$ time and takes $nH_0 + o(n \log |\Sigma|)$ bits of space. Golynski et al. [GMR06] improve the query bounds to $O(\log \log |\Sigma|)$ time, although they take more bits, namely, $n \log |\Sigma| + o(n \log |\Sigma|)$ of space. Nevertheless, their data structure presents the best query bounds for this problem.

Developing a dynamic text dictionary based on the wavelet structure can be done readily using dynamic bit dictionaries (as is done in [MN06]) since updates to a particular symbol s only affect $O(\log |\Sigma|)$ groups of symbols according to the hierarchical decomposition of the alphabet Σ. On the other hand, [GMR06] essentially treats each symbol in Σ individually; an update to symbol s could potentially affect *every* symbol in the alphabet. The only known solution to this problem is given by Makinen and Navarro [MN06], with an update/query bound of $O(\log n \log |\Sigma|)$. These bounds are far from optimal, especially in query-intensive settings.

In this paper, we develop a general framework that achieves amortized dynamic bounds and can dynamize any static text dictionary structure. Our approach collects all of the updates into a new data structure (called the onlyX structure in Section 3.4) and later merges this with a static text dictionary on the original text T. The interface and translation of positions from the static dictionary to our onlyX structure is handled by a suite of dynamic bit dictionaries. We use these structures to answer queries on

the fly; to do so, we combine the information we have stored to give the correct answer.

Using our framework, we can achieve near-optimal tradeoff for update/query times for the dynamic text dictionary problem. In particular, we achieve a dynamic text dictionary with $O((1/\epsilon)\log\log n)$ query time with an $O((1/\epsilon)n^{\epsilon})$ amortized update time. To the best of our knowledge, this is the best-known bound for the dynamic text dictionary problem.

Our emphasis on frequent queries and few updates is motivated by theoretical and practical consequences. Theoretically speaking, the lower bounds of [PD06] on prefix sums problem (which can be seen as a particular case of this problem) suggest that $O(\log\log n)$ time bounds (or any non-trivial sublogarithmic bound) cannot be achieved with just $O(\text{polylog}(n))$ update times. In practice, many XML indexing scenarios demand frequent query access, but the underlying data stays (relatively) static.

We list following contributions of this paper:

- We provide the first nearly-optimal result for the *dynamic text dictionary* problem on a dynamic text T. Our data structure requires $O((1/\epsilon)\log\log n)$ time to support $rank_s$, $select_s$, and *char* queries while supporting updates to the text T in amortized $O((1/\epsilon)n^{\epsilon})$ time. Our data structure is also compressed, taking just $n\log|\Sigma| + o(n\log|\Sigma|)$ bits of space
- Our results improve the query bounds of previous work, as well as highlight a near-optimal update/query tradeoff.
- Furthermore, our results provide a general framework to dynamize any static text dictionary with near-optimal update/query tradeoffs.
- We apply our dynamic results to the important problem of XML indexing. Using the XBW transform [FLMM05], we show how to perform navigational queries and insertion and deletion of paths (and subtrees) on an ordered tree T. We support these operations in $O((1/\epsilon\log\log n)$ time with an amortized update bound of $O((1/\epsilon)(n^{\epsilon} + h\log\log n))$ time, where h is the depth of the insertion or deletion in T. We also support the powerful $subpath(P)$ query in $O((m/\epsilon)\log\log n)$ time, where $m = |P|$.

1.1 Outline

In Section 2, we describe the RRR data structure [RRR02] and the static text dictionary of Golyinski et al. [GMR06] and some brief construction bounds. Section 3.2 describes our *BitIndel* data structure, which maintains a bitvector over insertions and deletions while supporting fast queries. Section 3.3 describes the first part of our dynamic text dictionary; we describe *inX*, which keeps track of where the original text T has been updated. In Section 3.4, we then describe *onlyX*, which actually stores the updates themselves. We conclude in Section 5.

2 Preliminaries

Suppose we are given a text T with n symbols drawn from an alphabet Σ. For $s \in \Sigma$, the following operations are useful in several applications.

- $T.rank_s(i)$ returns the number of symbols s up to position i in T;
- $T.select_s(\ell)$ returns the position of the ℓth symbol s in T;
- $T.char(\ell)$ returns the symbol s located in the ℓth position of T.

One important result for the case of bitvectors is [RRR02], which is a static bit dictionary supporting $rank_s$ and $select_s$ (and thus, *char*) queries in $O(1)$ time using $nH_0 + O(n\log\log n/\log n)$ bits of space. The RRR [RRR02] data structure can be constructed in $O(n)$ time. We summarize these important results in the following lemma.

Lemma 1 (RRR). *For a bitvector (i.e., $|\Sigma| = 2$) of length n, there exists a static data structure that supports rank, select, and char queries in $O(1)$ time using $nH_0 + O(n\log\log n/\log n)$ bits of space, while taking only $O(n)$ time to construct.*

Proof. The space bound and the query times follow directly from [RRR02]. For construction, a straightforward approach requires $O(n/\log n + t) = O(n)$ time at most, where t is the number of 1s in the bitvector. □

For the static text dictionary problem, Grossi et al. [GGV03] present a wavelet tree structure that answers queries in $O(\log|\Sigma|)$ time and takes $nH_0 + o(n)$ bits of space. In the special case where $|\Sigma| = O(\text{polylog}(n))$, [NFMM04] achieve $O(1)$ time for the queries. Golynski et al. [GMR06] improve the query bounds to $O(\log\log|\Sigma|)$ time, although they take more bits $n\log|\Sigma| + o(n\log|\Sigma|)$ of space. In this paper, we will make heavy use of GMR. We summarize its results below

Lemma 2 (GMR). *For a text T of length n drawn from alphabet Σ, there exists a static data structure that supports $select_s$ in $O(1)$ time and rank and char queries in $O(\log\log|\Sigma|)$ time using $n\log|\Sigma| + o(n\log|\Sigma|)$ bits of space space, while taking $O(n\log n)$ time to construct.*

Proof. The space and time bounds are discussed in [GMR06]. We defer the construction proof until the full version of the paper. □

2.1 Prefix-sum (PS) structure

Suppose we are given an non-negative integer array $A[1..t]$ such that $\sum_i A[i] \leq n$ for which we wish to devise a prefix sum structure. First, we calculate all the partial sums $P[i] = \sum_{j=1}^{i} A[i]$. We can regard P as a sorted array of prefix sums, such that $0 < P[i] \leq P[j] \leq n$ for all $i < j$. We describe a data structure based on van Emde Boas called PS that allows us to support the queries *sum* and *findsum* on P in $O(\log\log n)$ time using $O(t\log n)$ bits. We define these operations below:

- $sum(j)$ returns the partial sum $P[j]$;
- $findsum(i)$ returns the index j such that $sum(j) \leq i < sum(j+1)$.

We can construct this PS in $O(t)$ time. To support *sum*, we simply store array P explicitly, requiring $O(t\log n)$ bits of space.

To support *findsum*, we take the t prefix sums and cluster them into consecutive groups of size $O(\log n)$. Within a group, we use a balanced binary search tree to support *findsum* in $O(\log\log n)$ time in the standard way. Now we must determine which group to search for a given query. From each of the $O(t/\log n)$ groups, we store the largest prefix sum using a hashing implementation of a van Emde Boas (VEB) data structure. For the hashing, we use [Pagh, Theorem 1.1; Melhorn, Vishkin, Theorem A], so that we can construct the hash table deterministically in $O((t/\log n)\log n) = O(t)$ time and taking $O((t/\log n)\log n) = O(t)$ bits of space. Along with each entry in the hash table, we also store a pointer to its associated group to search further. To answer $findsum(i)$, we search the VEB structure to find the right group in $O(\log\log n)$ time. We then follow the pointer to the binary search tree and spend an additional $O(\log\log n)$ time.

3 Data structures

There are several data structures that support $rank_s$ and $select_s$ queries. They are broadly based on two different approaches: *logarithmic*, which create a binary search tree of height $\log|\Sigma|$ with each symbol's occurences stored in the leaves; and *log-logarithmic*, which are based on predecessor search and VEB. Despite the faster access of the log-logarithmic approach, it is difficult to update since each symbol $s \in \Sigma$ is treated separately and updates affect all other symbols. In contrast logarithmic approaches need only manage updates in a particular root-to-leaf path of their binary search tree, i.e., at most $O(\log|\Sigma|)$ internal nodes.

3.1 Overview of our data structure

Our solution is built with three main data structures:

- *BitIndel* bitvector supporting insertion and deletion;
- *StaticRankSelect* static structure supporting $rank_s$, $select_s$, and *char* on a text T;
- *DynamicRankSelect* dynamic rank and select structure taking more space than StaticRankSelect.

We use StaticRankSelect to maintain the original text T; we implement StaticRankSelect using GMR [GMR06] and merge updates with this structure every $O(n^{1-\epsilon}\log n)$ update operations. We keep track of the newly inserted symbols N in DynamicRankSelect and merge N with T as we have just described. Thus, DynamicRankSelect never contains more than $O(n^{1-\epsilon}\log n)$ symbols. We maintain DynamicRankSelect using $O(n^{1-\epsilon}\log^2 n) = o(n)$ bits of space. Finally, since merging N with T requires $O(n\log n)$ time, we arrive at an amortized $O((1/\epsilon)n^{\epsilon})$ time for updating these data structures. BitIndel is used to translate positions p_t from the old text T to the new positions p_n from the current text $\hat{T}$. (We maintain $\hat{T}$ implicitly through the use of BitIndel structures, StaticRankSelect, and DynamicRankSelect.)

3.2 Bit-vector dictionary with Indels: BitIndel

In this section, we describe a data structure (BitIndel) for a bitvector B of original length n that can handle insertions and deletions of bits anywhere in B while still supporting *rank* and *select* on the updated bitvector B' of length n'. Our structure supports these updates in $O(n^\epsilon)$ time and *rank* and *select* queries in $O((1/\epsilon)\log\log n)$ time. This is comparable to the problem considered by [MN06], where they support all of the above operations in $O(\log n)$ time.

Formally, we define the following update operations that we support on the current bitvector B' of length n':

- $insert_b(i)$ inserts the bit b in the ith position;
- $delete(i)$ deletes the bit located in the ith position;
- $flip(i)$ flips the bit in the ith position.

For bitvector B', we construct a B-tree ($\mathcal{T}$) with fanout between $[n^\epsilon, 2n^\epsilon]$. The leaves of $\mathcal{T}$ maintain contiguous chunks of B' ranging from $[n^\epsilon, 2n^\epsilon]$ in size, such that the ℓth (leftmost) leaf corresponds to the ℓth chunk of B'. Each leaf ℓ maintains an RRR [RRR02] data structure $\ell.R$ that answers *rank* and *select* queries on its $O(n^\epsilon)$-sized chunk in $O(1)$ time. Each internal node v of $\mathcal{T}$ maintains three arrays: $count_{\mathbf{0}}$, $count_{\mathbf{1}}$, and *size*. Let c_j denote the jth child node of v. The entry $count_{\mathbf{0}}[j]$ is the number of **0**s in the part of the bitvector in the subtree of c_j. The entry $count_{\mathbf{1}}[j]$ is the number of **1**s in the part of the bitvector in the subtree of c_j. The entry $size[j]$ is the total number of bits in the subtree of c_j. To have fast access to this information at each node, we build a PS structure on this information. (We don't actually store $count_{\mathbf{0}}$, $count_{\mathbf{1}}$, and *size* explicitly; rather, we store a PS structure for each array.)

The height of this tree is $O((1/\epsilon)\log_n n')$. To traverse down to a leaf for any operation, we use the PS structure at a node (using $O(\log\log n)$ time) to determine the next node to visit on the root-to-leaf path. Then, we query our RRR [RRR02] data structure $\ell.R$ at leaf ℓ and return the answer. Now we describe our operations in more detail. (Note that $rank_{\mathbf{0}}(i) = i - rank_{\mathbf{1}}(i)$.)

```
function v.rank_1(i) {
  if (leaf(v)) return v.R.rank_1(i);
  j ← v.size.findsum(i);
  return v.count_1.sum(j)+
    c_{j+1}.R.rank_1(i − v.size.sum(j));
}
```

```
function v.select_s(i) {
  if (leaf(v)) return v.R.select_s(i);
  j ← v.count_s.findsum(i);
  return v.size.sum(j)+
    c_{j+1}.R.select_s(i − v.count_s.sum(j));
}
```

Time Bounds. Each of the above queries requires $O(\log\log n)$ time per node traversed in the B-tree $\mathcal{T}$. Since there are at most $O((1/\epsilon)\log_n n')$ such nodes before encountering a leaf, the total time is $O((1/\epsilon)\log_n n')\log\log n)$.

Updates. The $flip(i)$ operation can be supported by performing a constant number of *insert*, *delete*, and *rank* operations. At every update operation, we traverse the B-tree as before. The prefix-sum data structures in each internal node along the path are rebuilt in $O(n^\epsilon)$ time per node. At the leaf, R is rebuilt. If the leaf node manages more than $2n^\epsilon$ symbols or less than n^ϵ, we invoke the standard B-tree merge/split routines, propagating them up the tree as appropriate. In the worst case, updates take $O((1/\epsilon)n^\epsilon\log_n n')$. The amortized time is easily bounded by $O((1/\epsilon)(\log_n n' + n^\epsilon))$. Furthermore, we rebuild the entire data structure every $O(n^{1-\epsilon})$ updates.

Space. There are at most $O(n'/n^{2\epsilon})$ internal nodes, each taking $O(n^\epsilon\log n')$ bits bits. Thus, the total space for the internal nodes is $O((n'/n^\epsilon)\log n')$. Let $n_{\mathbf{1}}$ be the number of **1**s in B'. The space for the bottom-level R structures can be bounded by $\lceil\log\binom{n'}{n_{\mathbf{1}}}\rceil + o(n')$ bits.

Lemma 3. *Given a bitvector B' with length n' and original length n, we can create a data structure that takes $O((n'/n^\epsilon)\log n') + \lceil\log\binom{n'}{n_{\mathbf{1}}}\rceil + o(n')$ bits and supports rank and select in $O(\log\log n))$ time, and indel in $O((1/\epsilon)(\log_n n' + n^\epsilon))$ amortized time.* □

3.3 Insert-X-Delete-any: inX

Let `x` and `d` be symbols other than those in alphabet Σ. In this section, we describe a data structure on a text T of length n supporting $rank_s$ and $select_s$ that can handle $delete(i)$ and $insert_{\mathtt{x}}(i)$. Notice that insertions and deletions affect the answers returned for symbols in the alphabet Σ. For example, T may be `abcaab`, where $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Here, $rank_{\mathtt{a}}(4) = 2$ and $select_{\mathtt{a}}(3) = 5$. Let $\hat{T}$ be the current text after some number of insertions and deletions of symbol `x`. Initially, $\hat{T} = T$. After some insertions, the current $\hat{T}$ may be `axxxbcaxabx`. Notice that $rank_{\mathtt{a}}(4) = 1$ and $select_{\mathtt{a}}(3) = 9$. We represent $\hat{T}$ by the text T',

in which the symbols from the original text T are never deleted, but are instead replaced by a special symbol d. Continuing the example, after some deletions of symbols from T, T' may be axxxddaxabx. Notice that $rank_{\mathtt{a}}(4) = 1$ and $select_{\mathtt{a}}(3) = 7$.

We define an *insert vector* I such that $I[i] = \mathbf{1}$ if and only if $T'[i] = \mathtt{x}$. Similarly, we define a *delete vector* D such that $D[i] = \mathbf{1}$ if and only if $T'[i] = \mathtt{d}$. We also define a delete vector D_s for each symbol s such that $D_s[i] = \mathbf{1}$ if and only if the ith s in the original text T was deleted. The text T' is merely a conceptual text: we refer to it for ease of exposition but we actually maintain $\hat{T}$ instead.

To store $\hat{T}$, we store T using the StaticRankSelect data structure and store all of the I, D, D_s bitvectors using the BitInDel structure. Now, we show how to perform $\hat{T}.insert_{\mathtt{x}}(i), \hat{T}.delete(i), \hat{T}.rank_s(i)$ and $\hat{T}.select_s(i)$:

$\hat{T}.insert_{\mathtt{x}}(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. Then we must update our various vectors. We perform $I.insert_{\mathbf{1}}(i')$ on our insert vector, and $D.insert_{\mathbf{0}}(i')$ on our delete vector.

$\hat{T}.delete(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If i' is newly-inserted (i.e., $I[i'] = \mathbf{1}$), then we perform $I.delete(i')$ and $D.delete(i')$ to reverse the insertion process from above. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Let $s = T.char(i'')$. Finally, to delete the symbol, we perform $D.flip(i')$ and $D_s.flip(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.rank_s(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $s = \mathtt{x}$, return $I.rank_{\mathbf{1}}(i')$. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Finally, we return $D_s.rank_{\mathbf{0}}(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.select_s(i)$. If $s = \mathtt{x}$, compute $j = I.select_{\mathbf{1}}(i)$ and return $D.rank_{\mathbf{0}}(j)$. Otherwise, we compute $k = D_s.select_{\mathbf{0}}(i)$ to determine i's position among the s symbols from T. We then compute $k' = T.select_s(k)$ to determine its original position in T. Now the position k' from T needs to be mapped to its appropriate location in $\hat{T}$. Similar to the first case, we perform $k'' = I.select_{\mathbf{0}}(k')$ and return $D.rank_{\mathbf{0}}(k'')$, which corresponds to the right position of $\hat{T}$.

$\hat{T}.char(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $I[i'] = 1$, return x. Otherwise, we convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$ and return $T.char(i'')$.

Space and Time. As can be seen, each of the *rank* and *select* operations requires a constant number of accesses to BitIndel and StaticRankSelect structures, thus taking $O((1/\epsilon)(\log_n n') \log\log n)$ time to perform. The *indel* operations require $O(n^\epsilon)$ update time, owing to the BitIndel data structure. The space required for the above data structures comes from the StaticRankSelect structure, which requires $O(n \log|\Sigma| + o(n \log|\Sigma|))$ bits of space, and the many BitIndel structures, whose space can be bounded by $\log \binom{n'}{n} + 2\log \binom{n'}{n''} + o(n')$ where n'' is number of deletes.

Theorem 1. *Given a text T of length n drawn from an alphabet Σ, we create a data structure that takes $n\log|\Sigma + o(n\log|\Sigma|) + \log\binom{n'}{n} + 2\log\binom{n'}{n''} + o(n')$ bits of space and supports $rank_s(i)$ and $select_s(i)$ in $O((1/\epsilon)(\log_n n')\log\log n)$ time. We can also support $insert_{\mathtt{x}}(i)$ and $delete_s(i)$ in $O((1/\epsilon)n^\epsilon)$ time. If n'' and $n' - n$ are less than $n^{1-\epsilon}$, we require $n\log|\Sigma| + o(n\log|\Sigma|) + o(n)$ bits of space.*

3.4 onlyX-structure

In this section, we describe a data structure for maintaining a dynamic array of symbols that supports $rank_s$ and $select_s$ queries in $O((1/\epsilon)(\log_n n')\log\log n)$ time, for any ϵ with $0 < \epsilon < 1$; here, we assume that the maximum number of symbols in the array is n. Our data structure takes $O(n'\log n)$ bits, where n' is the current number of symbols; for each update (i.e., insertion or deletion of a symbol), it can be done in amortized $O((1/\epsilon)n^\epsilon)$ time.

In the following, we first review a previous data structure result called the Weight Balanced B-tree (WBB tree) that was also used in [RRR01, HSS03]. Then, we show that our data structure can be implemented by a simple instantiation of the WBB tree.

3.4.1 Weight Balanced B-tree (WBB Tree)

We define a weight balanced B-tree as follows: all leaves of the WBB tree are considered to be at level 0. A level-i node is connected to its parent node at level $i + 1$. We define a *weight-balance condition*, such

that for any node v at level i, the number of leaves in v's subtree is between $0.5b^i + 1$ and $2b^i - 1$, where b is the fanout factor. Thus, the degree of an internal node is $\Theta(b)$ (from b to $4b$), such that the height of the tree is $\Theta(\log_b n')$, where n' is the number of leaves in the current tree.

After a leaf is inserted into the tree, the weight-balance condition of some level-i ancestor of the leaf, say v, may be violated. Precisely, this case happens when the number of leaves in v's subtree is $2b^i$. In this case, v will be split into two new nodes at the same level (called a *split* operation), each of them becoming the root of a perfect subtree with b^i leaves. (This split could cause a restructuring of the entire subtree that was split, but this follows standard techniques.)

On the other hand, in case a leaf is deleted, the weight-balance condition of v at level i may be violated; that is, the number of leaves in v's subtree becomes $0.5b^i$. In this case, v is merged with one of its neighboring siblings, and there will be two cases:

(i) if the total number of leaves after merging is less than $1.5b^i$, the update finishes (called a *merge* operation);

(ii) otherwise, the merged node is further split into two nodes, each of them becoming the root of a subtree with half the number of leaves (called a *merge-then-split* operation).

Based on the above updating process, we have the following lemma and corollary.

Lemma 4. *Except the root, when a node v at level i violates the weight-balance condition, at least $\Theta(b^i)$ leaves are inserted or deleted v's subtree since the creation of v.*

Proof. A node is created when there is either a split, merge, or merge-then-split event. As a result, node v contains at least $0.75b^i$ leaves (by merge-then-split) and at most $1.5b^i$ leaves at its creation. Thus, at least $0.25b^i$ leaves are deleted or at least $0.5b^i$ leaves are inserted before v can violate the weight-balance condition. □

Corollary 1. *Suppose that c_i is the maximum cost of a split, a merge, or a merge-then-split operation when a level-i node violates the weight-balance condition. The amortized cost for supporting the above operations due to an insertion or deletion of a leaf is at most $\Theta(\sum_{i=1}^{h} c_i/b^i)$, where h denotes the current height of the tree.*

Proof. We prove this result by a simple accounting method. A node is created with zero tokens; when a leaf is inserted or deleted, it gives each of its level-i ancestors $\Theta(c_i/b^i)$ tokens (precisely, $4c_i/b^i$ tokens for deletion and $2c_i/b^i$ tokens for insertion). Thus, the total number of tokens given is $\Theta(\sum_{i=1}^{h} c_i/b^i)$ during an insertion or deletion operation. It is easy to verify that there are at least c_i tokens when a node at level i violates the weight-balance condition. In other words, an amortized cost of $\Theta(\sum_{i=1}^{h} c_i/b^i)$ for leaf insertion or deletion is enough to support split, merge, or merge-then-split operations. □

3.4.2 Dynamic Rank-Select Structure based on WBB Tree

Let T be the dynamic text that we want to maintain, and where symbol of T is drawn from alphabet Σ Let n' be the length of T, and we assume that n' is never more than some pre-defined value n.

We describe how to apply the WBB Tree to maintain T while supporting $rank_s$ and $select_s$ efficiently, for any $s \in \Sigma$. In particular, we choose $\epsilon < 1$ and store the symbols of T in a WBB W with fanout factor $b = n^\delta$ where $\delta = \epsilon/2$ such that the ith (leftmost) leaf of W stores $T[ib]$. Each node at level 1 will correspond to a substring of T with $O(b)$ symbols, and we will maintain a GMR-structure for that substring so that $rank_s$ and $select_s$ are computed for that substring in $O(\log\log|\Sigma|)$ time. In each level-ℓ node v_ℓ with $\ell \geq 2$, we store an array *size* such that $size[i]$ stores the number of symbols in the subtree of its ith (leftmost) child. To have fast access to this information at each node, we build a PS structure to store *size*. Also, for each symbol s that appears in the subtree of v_ℓ, v_ℓ is associated with an *s-structure*, which consists of three arrays:

- pos_s: $pos_s[i]$ stores the index of v_ℓ's ith leftmost child whose subtree contains s;[1]
- num_s: $num_s[i]$ stores the number of s in v_ℓ's ith leftmost child whose subtree contains s;[3]
- ptr_s: $ptr_s[i]$ stores a pointer to the s-structure of v_ℓ's ith leftmost child whose subtree contains s.

[1] For example, if the 2nd, 4th, 5th, and 7th children are the only children of v_ℓ whose subtree contains s, we have $pos_s[1] = 2, pos_s[2] = 4, pos_s[3] = 5, pos_s[4] = 7$.

[3] Continuing with the example for pos_s, if the 2nd, 4th, 5th, and 7th children of v_ℓ contain respectively 11, 23, 4, and 6 occurrences of s in their subtrees, we have $num_s[1] = 11, num_s[2] = 23, num_s[3] = 4, num_s[4] = 6$.

The arrays in each s-structure ($size_s$, pos_s, and num_s) are stored using a PS data structure so that we can support $O(\log\log n)$-time *sum* and *findsum* queries in $size_s$ or num_s, and $O(\log\log n)$-time *rank* and *select* queries in pos_s. (These *rank* and *select* operations are analagous to *sum* and *findsum* queries, but we refer to them as *rank* and *select* for ease of exposition.) The list ptr_s is stored in a simple array.

We also maintain another B-tree B with fanout n^δ such that each leaf ℓ_s corresponds to a symbol s that is currently present in the text T. Each leaf stores the number of (nonzero) occurrences of s in T, along with a pointer to its corresponding s-structure in the root of W. The height of B is $O(\log_{n^\epsilon}|\Sigma|) = O(1/\epsilon)$, since we assume $|\Sigma| \le n$.

Answering *char*(i). We can answer this query in $O((1/\epsilon)\log\log n)$ time by maintaining a B-tree with fanout $b = n^\delta$ over the text. We call this tree the *text B-tree*.

Answering *rank*$_s(p)$. Recall that $rank_s(p)$ tells the number of occurrences of s in $T[1..p]$. We first query B to determine if s occurs in T. If not, return 0. Otherwise, we follow the pointer from B to its s-structure. We then perform $r.size_s.findsum(p)$ to determine the child c_i of root r from W that contains $T[p]$. Suppose that $T[p]$ is in the subtree rooted at the ith child c_i of r. Then, $rank_s$ consists of two parts: the number of occurrences $m_1 = r.num_s.sum(j)$ (with $j = r.pos_s.rank(i-1)$) in the first $i-1$ children of r, and m_2, the number of occurrences of s in c_i. If $r.pos_s.rank(i) \neq j+1$ (c_i contains no s symbols), return m_1. Otherwise, we retrieve the s-structure of c_i by its pointer $r.ptr[j+1]$ and continue counting the remaining occurrences of s before $T[p]$ in the WBB tree W. We will eventually return $m_1 + m_2$.

The above process either (i) stops at some ancestor of the leaf of $T[p]$ whose subtree does not contain s, in which case we can report the desired rank, or (ii) it stops at the level-1 node containing $T[p]$, in which case the number of remaining occurrences can be determined by a $rank_s$ query in the GMR-structure in $O(\log\log|\Sigma|)$ time. Since it takes $O(\log\log n/\epsilon)$ time to check the B-tree B at the beginning, and it takes $O(\log\log n)$ time to descend each of the $O(1/\epsilon)$ levels in the WBB-tree to count the remaining occurrences, the total time is $O(\log\log n/\epsilon)$.

Answering *select*$_s(j)$. Recall that $select_s(j)$ tells the number of symbols (inclusive) before the jth occurrence of s in T. We follow a similar procedure to the above procedure for $rank_s$. We first query B to determine if s occurs at least j times in T. If not, we return -1. Otherwise, we discover the ith child c_i of root r from W that contains the jth s symbol. We compute $i = r.pos_s.select(r.num_s.findsum(j))$ to find out c_i.

Then, $select_s$ consists of two parts: the number of symbols $m_1 = r.size.sum(i)$ in the first $i-1$ children of r, and m_2, the number of symbols in c_i before the jth s. We retrieve the s-structure of c_i by its pointer $r.ptr[r.num_s.findsum(j)]$ and continue counting the remaining symbols on or before the jth occurrence of s in T. We will eventually return $m_1 + m_2$. The above process will stop at the level-1 node containing the jth occurrence of s in T, in which case the number of symbols on or before it maintained by this level-1 node can be determined by a $select_s$ query in the GMR-structure in $O(\log\log|\Sigma|)$ time. With similar time analysis as in $rank_s$, the total time is $O(\log\log n/\epsilon)$.

Updates. We can update the text B-tree in $O((1/\epsilon)n^\epsilon)$ time. We use a naive approach to handle updates due to the insertion or deletion of symbols in T: For each list in the WBB-tree and for each GMR-structure that is affected, we rebuild it from scratch. In the case that no split, merge, or merge-then-split operation occurs in the WBB-tree, an insertion or deletion of s at $T[p]$ will affect the GMR-structure containing $T[p]$, and two structures in each ancestor node of the leaf containing $T[p]$: the *size* array and the s-structure corresponding to the inserted (deleted) symbol. The update cost is $O(n^\delta\log n) = O(n^\epsilon)$ for the GMR-structure and for each ancestor, so in total it takes $O((1/\epsilon)n^\epsilon)$ time.

If a split, merge, or merge-then-split operation occurs at some level-ℓ node v_ℓ in the WBB-tree, we need to rebuild the *size* array and s-structures for all newly created nodes, along with updating the *size* array and s-structures of the parent of v_ℓ. In the worst case, it requires $O(n^{(\ell+1)\epsilon}\log n)$ time. By Corollary 1, the amortized update takes $O(n^\epsilon/\epsilon)$ time.

In summary, each update due to an insertion or deletion of symbols in T can be done in amortized $O(n^\epsilon/\epsilon)$ time.

Space complexity. The space for the text B-tree is $O(n\log|\Sigma| + n^{1-\epsilon}\log n)$ bits. The total space of all $O(n^{1-\epsilon})$ GMR-structures can be bounded by $O(n\log|\Sigma|)$ bits. The space for the B-tree B (maintaining distinct symbols in T) is $O(|\Sigma|\log n)$ bits. The total number of words to store all arrays in the internal nodes is linear to the total number of entries, so the total space for these arrays is $O(n\log n/\epsilon)$ bits. (In particular, each of the n symbols from the text T contributes $O((1/\epsilon)\log n)$ bits of space to maintain

information about itself; in total, the bound is as above.) In summary, the total space of the above dynamic rank-select structure is $O(n \log n/\epsilon)$ bits.

Summarizing the above discussions, we conclude this section by the following theorem.

Theorem 2. *For a dynamic text T of length at most n, we can maintain a data structure on T to support $rank_s$, $select_s$, and $char$ $O(\log \log n/\epsilon)$ time, and insertion/deletion of a symbol in amortized $O(n^\epsilon/\epsilon)$ time. The space of the date structure is $O(n \log n/\epsilon)$ bits.* □

3.5 The final data structure

Here we describe our final structure, which supports insertions and deletions of any symbol. To do this, we maintain two structures: our inX structure on $\hat{T}$ and the onlyX structure, where all of the new symbols are actually inserted and maintained. After every $O(n^{1-\epsilon} \log n)$ update operations, the onlyX structure is merged into the original text T and a new T is generated. All associated data structures are also rebuilt. Since this construction process could take at most $O(n \log n)$ time, this cost can be amortized to $O((1/\epsilon)n^\epsilon)$ per update. The StaticRankSelect structure on T takes $n \log |\Sigma| + o(n \log |\Sigma|)$ bits of space. With this frequent rebuilding, all of the other supporting structures take only $o(n)$ bits of space.

We augment the above two structures with a few additional BitIndel structures. In particular, for each symbol s, we maintain a bitvector I_s such that $I_s[i] = \mathbf{1}$ if and only if the ith occurence of s is stored in the onlyX structure. With the above structures, we quickly describe how to support $rank_s(i)$ and $select_s(i)$.

For $rank_s(i)$, we first find $j = inX.rank_s(i)$. We then find $k = inX.rank_{\mathbf{x}}(i)$ and return $j + onlyX.rank_s(k)$. For $select_s(i)$, we first find whether the ith occurence of c belongs to the inX structure or the onlyX structure. If $I_s[i] = \mathbf{0}$, this means that the ith item is one of the original symbols from T; we query $inX.select_s(j)$ in this case, where $j = I_s.rank_{\mathbf{0}}(i)$. Otherwise, we compute $j = I_s.rank_{\mathbf{1}}(i)$ to translate i into its corresponding position among new symbols. Then, we compute $j' = onlyX.select_s(j)$, its location in $\hat{T}$ and return $inX.select_x(j')$.

Finally, we show how to maintain I_s during updates. For $delete(i)$, compute $\hat{T}[i] = s$. We then perform $I_s.delete(inX.rank_s(i))$. For $insert_s(i)$, after inserting s in $\hat{T}$, we insert it into I_s by performing $I_s.insert_{\mathbf{1}}(inX.rank_s(i))$. Let $n_{\mathbf{x}}$ be the number of symbols stored in the onlyX structure. We can bound the space for these new BitIndel data structures using RRR [RRR02] and Jensen's inequality by $\lceil \log \binom{n'}{n_{\mathbf{x}}} \rceil + o(n') = O(n^{1-\epsilon} \log^2 n) + o(n) = o(n)$ bits of space. Thus, we arrive at the following theorem.

Theorem 3. *Given a text T of length n drawn from an alphabet Σ, we create a data structure that takes $n \log |\Sigma + o(n \log |\Sigma|) + o(n)$ bits of space and supports $rank_s(i)$, $select_s(i)$, and $char(i)$ $O((1/\epsilon) \log \log n + \log \log |\Sigma|)$ time and $insert(i)$ and $delete(i)$ updates in $O((1/\epsilon)n^\epsilon)$ time.* □

4 XBW and Dynamic XML Indexing

In this section, we describe an application of our dynamic multi-symbol rank/select data structure to dynamizing the XBW transform [FLMM05] for an arbitrary ordered tree T where each of the n nodes in T has a label drawn from alphabet Σ. To ease our notation, we will also number our symbols from $[0, |\Sigma| - 1]$ such that the sth symbol is also the sth lexicographically-ordered one. We'll call this symbol s. Our dynamic XBW structure supports several operations in T

- $v.insert(P)$, which inserts the path P at node v;
- $v.delete()$, which removes the root-to-v path for a leaf v
- $subpath(P)$, which finds all occurrences of the path P;
- $v.parent()$, returns the parent node of v in T;
- $v.child(i)$, returns the ith child node of v;
- $v.child(s)$, returns any child node of v labeled s.

Before explaining our data structure, we first give a brief description of the XBW transform [FLMM05]. For a node v in T, let $\ell[v] = \mathbf{1}$ if and only if v is the rightmost child of its parent in T. Let $\alpha[v]$ be the label of v, and $\pi[v]$ be the string obtained by concatenating the labels on the upward path from $v.parent()$ to the root of T. We further assume that the node labels can be separated into two disjoint sets Σ_i and Σ_l of labels for internal nodes and leaves (respectively). We also let n_i be the number of internal nodes of T and n_ℓ be the number of leaves of T. We then construct a set S of n triplets, one for each tree node:

- Visit T in pre-order. For each visited node v add the triplet $s[v] = \langle \ell[v], \alpha[v], \pi[v] \rangle$ into S;
- Stable-sort S according to the π component of each triple.

The (output of the) XBW transform consists of the arrays S_ℓ and S_α, where these refer to the first and second components of each triplet (respectively) after the stable sort has been performed. Ferragina, et al show in [?] that the tree T can be reconstructed by storing these arrays. The above transform is reminiscent of the Burrows-Wheeler Transform (BWT) for text documents. Their structure supports navigational queries (*parent*, *child*) operations, as well as a *subpath*(P) search, which finds the nodes v such that the reversed path $rev(P)$ is a prefix of the concatenated string $\alpha[v]\pi[v]$. In summary, they achieve the following theorem for the static ordered trees T:

Theorem 4 (Static XBW REF). *For any ordered tree T with node labels drawn from an alphabet Σ, there exists a static succinct representation of it using the XBW transform [FLMM05] that takes at most $nH_0\langle S_\alpha \rangle + 2n + o(n)$ bits of space, while supporting navigational queries in $O(\log |\Sigma|)$ time. The representation can also answer a subpath(P) query in $O(m \log |\Sigma|)$ time, where m is the length of path P.* □

The full details of the result can be found in [FLMM05]. Here, we briefly recap the data structures used in their solution. For our result, we will show that replacing these structures with their dynamic counterpart is sufficient to achieve a powerful facility to update ordered trees (such as XML trees). For S_ℓ, [FLMM05] use an RRR [RRR02] data structure to maintain the bitvector of length n containing n_i **1**s in $\log \binom{n}{n_i} + o(n)$ bits of space. For S_α, [FLMM05] keep two data structures: F, a structure that keeps track of the number of occurences of each symbol s in Σ. F is (conceptually) a bitvector of length $n + |\Sigma|$ storing $|\Sigma|$ **1**s such that $select_{\mathbf{1}}(i) - select_{\mathbf{1}}(i-1) - 1$ indicates the number of occurrences of the ith label s in T. Finally, S_α is stored using a wavelet tree [GGV03].

For our dynamic XBW data structure, we replace the static implementations of S_ℓ and F with our BitIndel data structure, supporting *rank* and *select* in $O(\log \log n)$ time and updates in $O((1/\epsilon) \log_n n' + n^\epsilon)$ amortized time. Then, we replace the S_α data structure with our "final structure" that allows $rank_s$ and $select_s$ in $O((1/\epsilon) \log \log n)$ time and supports insertions and deletions in $O((1/\epsilon) n^\epsilon)$ time. We use the same algorithms for *parent* and *child* operations as [FLMM05]. Since these algorithms require a constant number of queries to the above data structures, we can now support these operations in $O((1/\epsilon) \log \log n)$ time. For *subpath*(P), we again use the same algorithm, taking $O((m/\epsilon) \log \log n)$ time, where m is the length of P.

For *insert*(P) and *delete*(), these operations will be defined on the original tree T for some node u where we want to begin inserting or deleting. We describe a method to translate any node u into a corresponding position v such that the triplet $S[v]$ in the XBW transform [FLMM05] corresponds to node u in T. For a path from root r to a node u in T, say $P = (u_0, u_1, u_2, \cdots, u_{h-1}, u_h)$ with $u_0 = r$ and $u_h = u$, we describe a sequence of child indices $C_u = c_1 c_2 \ldots c_h$, where c_i indicates that u_i is the c_ith child of u_{i-1}. To translate u into the corresponding position v in the XBW transform [FLMM05], we perform the following *convert* operation.

```
function convert(C_u) {
  v ← 1; // v is the root
  for (i = 1; i ≤ h; i++)
    v ← v.child(c_i);
  return v;
}
```

The above operation takes $O((h/\epsilon) \log \log n)$ time to perform with our dynamic data structures, where $h + 1$ is the depth of the node to be modified. Our later operations will take this much additional time. We state the following lemma.

Lemma 5. *For any node u at depth $h+1$ in tree T, we can find its corresponding position in the XBW transform [FLMM05] in $O(ht(n))$ time, where $t(n)$ is the amount of time taken by a data structure storing the XBW transform to perform a child(i) navigational operation.* □

We now describe how to support $v.insert(P)$ and $v.delete()$ for node v in the XBW transform [FLMM05]. For convenience, we rewrite $P = p_1 p_2 \cdots p_m$ as the concatenation of its m symbols. Furthermore, we assume that node v refers to its position in the XBW transform (easily done with $convert(c_v)$). For $v.insert(P)$, we begin at v and find v's last child. We then insert the next symbol in P after this child,

making the appropriate changes to S_ℓ and S_α. We also update F so that it maintains the correct count of alphabet symbols. For $v.delete()$, note that it's sufficient to simply know the leaf node $l = v$ of the path we wish to delete. To execute a deletion, we remove this leaf l and propagate to l's parent, making the appropriate changes to F, S_ℓ, and S_α. We terminate if l's parent has more than one child.

The above process can be expanded to also include routines for subtree insertion and deletion (*tinsert*, *tdelete*). Notice that the above algorithms require $O(m)$ queries to our dynamic data structures to insert or delete a path of length m. Thus, we arrive at the following theorem.

Theorem 5 (Dynamic XBW). *For any ordered tree T, there exists a dynamic succinct representation of it using the XBW transform [FLMM05] that takes at most $n \log |\Sigma| + o(n \log |\Sigma|) + 2n$ bits of space, while supporting navigational queries in $O((1/\epsilon) \log \log n)$ time. The representation can also answer a subpath(P) query in $O((m/\epsilon) \log \log n)$ time, where m is the length of path P. The update operations insert(P) and delete() at node u for this structure take $O((1/\epsilon)(n^\epsilon + h \log \log n))$ amortized time, where h is the depth of node u in T.* □

5 Conclusions and Implications of Our Result

We conclude with following discussion on results that can be readily obtained by tweaking our framework. We show many instances where our results are nearly tight against the previously best-known results. Some of our observations are results of independent interest; however, in the interest of maintaining a focused exposition, we defer the detailed description of these results to the full paper.

Memory Allocation Issues. As with any space-compact dynamic data structure, there are issues with memory allocation and fragmentation. In the results we describe in this paper, we only count the space that is actively used by the data structure: We do not count the wasted space due to memory fragmentation. However, this additional space overhead can be bounded by $o(n)$ bits if we manage memory in pages containing $n^{\epsilon/2}$ items. In this case, the space required for the virtual memory translation table can also be bounded by $o(n)$ bits.

$O(1)$ Query Time BitIndel. In this paper, we have only described a BitIndel data structure that takes $O((1/\epsilon) \log \log n)$ time to answer queries, since this was sufficient to achieve our final result. However, we can modify BitIndel to perform $O(1)$ query time by taking three times as much space, i.e., $3nH_0 + o(n)$ bits. We briefly describe how this is done.

Instead of a single B-tree, we store three WBB trees, weight balanced by *size*, $count_0$, and $count_1$. For the partial sum problem, $O(1)$ query time can be achieved if each array entry $A[i]$ is between x and $2x$ for some non-negative integer x [HSS03]. *rank* queries can be answered using the WBB for *size*, while $select_s$ can be answered with the WBB for $count_s$. Since the size of all these BitIndel structures is strictly $o(n)$ in our main structure, the space bound doesn't change. Still, despite such a BitIndel structure, the main bottleneck on time is in the onlyX structure, where we still need $O(\log \log n)$ time.

Special Cases of our BitIndel Framework. If we change our BitIndel structure such that the bottom-level RRR [RRR02] data structures are built on $[\log^2 n, 2 \log^2 n]$ bits each and set the B-tree fanout factor $b = 2$, we can obtain $O(\log n)$ update time with $O(\log n)$ query time. Thus, our BitIndel data structure is a generalization of [MN06].

Alternatives to GMR [GMR06]. Our choice to use the GMR structure to store StaticRankSelect was due to its best known query times. We present two cases where alternative choice leads to interesting results:

- To achieve entropy compression, we use the wavelet tree [GGV03] instead of [GMR06] and get query times of $O(\log |\Sigma| + \log \log n)$.
- When $|\Sigma| = O(\text{polylog}(n))$, we can achieve $O(\log \log n)$ query time by using [NFMM04].

Tightness of Our Result. For the case when $|\Sigma| = O(\text{polylog}(n))$, we can modify the OnlyX structure by using separate select structures for each symbol s to achieve $O(1)$ queries. This modification is similar to the one we made for our $O(1)$ BitIndel structure. In this case, our space becomes $O(|\Sigma| n \log n/\epsilon)$ and each update has to be carried out in all of the $|\Sigma|$ structures, thus taking $O((1/\epsilon) n^\epsilon |\Sigma|)$ time for updates.

When $|\Sigma| = O(\text{polylog}(n))$, the space overhead is still $o(n)$ and the update time can still be considered $O(n^\epsilon)$. Using [NFMM04], we now have $O(1)$ query time for StaticRankSelect, and thus an overall $O(1)$ query time. When $|\Sigma| = \Omega(n^\epsilon)$ our $O((1/\epsilon) \log \log n)$ bound is equivalent to the best known static bound of $O(\log \log |\Sigma|)$ given by GMR.

References

[BM99] A. Brodnik and I. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, October 1999.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[FLMM06] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *Proceedings of the International World Wide Web Conference*, 2006.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 41. IEEE, 2000.

[FM01] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, January 2001.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 32, May 2000.

[HMP00] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. Submitted manuscript (`http://www.daimi.au.dk/~bromille/Papers/det-jour.pdf`), 2000.

[HSS03] W. K. Hon, K. Sadakane, and W. K. Sung. Succinct data structures for searchable partial sums. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 505–516, 2003.

[Jac89] G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, January 1989.

[MN06] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proceedings of the Symposium on Combinatorial Pattern Matching*, pages 306–317, 2006.

[NFMM04] G. Navarro, P. Ferragina, G. Manzini, and V. Mäkinen. Succinct representation of sequences. Technical report, University of Chile, Technical Report TR/DCC-2004-5, Department of Computer Sciences, 2004.

[Pag99] R. Pagh. Low redundancy in static dictionaries with $O(1)$ worst case lookup time. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604. Springer-Verlag, 1999.

[PD06] M. Patrascu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[RRR01] R. Raman, V. Raman, and S. Rao. Succinct dynamic data structures. pages 426–437, 2001.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[SG06] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 1230–1239, 2006.