Dynamic Data Structures for Document Collections and Graphs

J. Ian Munro
Cheriton School of Computer Science
University of Waterloo
imunro@uwaterloo.ca.

Yakov Nekrich
Cheriton School of Computer Science
University of Waterloo
ynekrich@uwaterloo.ca.

Jeffrey Scott Vitter
Department of Electrical Engineering & Computer Science
University of Kansas
jsv@ku.edu.

ABSTRACT

In the dynamic indexing problem, we must maintain a changing collection of text documents so that we can efficiently support insertions, deletions, and pattern matching queries. We are especially interested in developing efficient data structures that store and query the documents in compressed form. All previous compressed solutions to this problem rely on answering rank and select queries on a dynamic sequence of symbols. Because of the lower bound in [Fredman and Saks, 1989], answering rank queries presents a bottleneck in compressed dynamic indexing. In this paper we show how this lower bound can be circumvented using our new framework. We demonstrate that the gap between static and dynamic variants of the indexing problem can be almost closed. Our method is based on a novel framework for adding dynamism to static compressed data structures. Our framework also applies more generally to dynamizing other problems. We show, for example, how our framework can be applied to develop compressed representations of dynamic graphs and binary relations.

Categories and Subject Descriptors

F2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Sorting and searching; H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing—Indexing methods

Keywords

Compressed Data Structures; Text Indexes; Graph Data Structures

1. INTRODUCTION

Motivated by the preponderance of massive data sets (so-called “big data”), it is becoming increasingly useful to store data in compressed form and moreover to manipulate and query the data while in compressed form. For that reason, such compressed data structures have been developed in the context of text indexing, graph representations, XML indexes, labeled trees, and many other applications. In this paper we describe a general framework to convert known static compressed data structures into dynamic compressed data structures. We show how this framework can be used to obtain significant improvements for two important dynamic problems: maintaining a dynamic graph and storing a dynamic collection of documents. We expect that our general framework will find further applications.

In the indexing problem, we keep a text or a collection of texts in a data structure, so that, given a query pattern, we can list all occurrences of the query pattern in the texts. This problem is one of the most fundamental in the area of string algorithms. Data structures that use $O(n \log n)$ bits of space can answer pattern matching queries in optimal time $O(|P| + occ)$, where $|P|$ denotes the length of the query pattern $P$ and occ is the number of occurrences of $P$. Because of the large volumes of data stored in text databases and document collections, we are especially interested in data structures that store the text or texts in compressed form and at the same time can answer pattern matching queries efficiently. Compressed indexing problem was extensively studied in the static scenario and during the last two decades significant progress has been achieved; we refer to a survey [30] for an overview of previous results in this area.

In the dynamic indexing problem, also known as the library management problem, we maintain a collection of documents (texts) in a data structure under insertions and deletions of texts. It is not difficult to keep a dynamic collection of texts in $O(n)$ words (i.e., $O(n \log n)$ bits) and support pattern matching queries at the same time. For instance, we can maintain suffixes of all texts in a suffix tree; when a new text is added or deleted, we add all suffixes of the new text to the suffix tree (respectively, remove all suffixes of the deleted text from the suffix tree). We refer the reader to the full version of this paper [29] for a more detailed description of the $O(n \log n)$-bit solution. The problem of keeping a dynamic document collection in compressed form is however more challenging. Compressed data structures for the library management problem were considered in a number of papers [9, 25, 8, 25, 23, 26, 15, 24, 16, 20, 32, 31]. In spite of previous work, the query times of previously described dynamic data structures significantly exceed the query times of the best static indexes. In this paper we show that the gap between the static and the dynamic variants
of the compressed indexing problem can be closed or almost closed. Furthermore we show that our approach can be applied to the succinct representation of dynamic graphs and binary relations that supports basic adjacency and neighbor queries. Again our technique significantly reduces the gap between static and dynamic variants of this problem.

These problems arise often in database applications. For example, reporting or counting occurrences of a string in a dynamic collection of documents is an important operation in text databases and web browsers. Similar tasks also arise in data analytics. Suppose that we keep a search log and want to find out how many times URLs containing a certain substring were accessed. Finally the indexing problem is closely related to the problem of substring occurrence estimation [33]. The latter problem is used in solutions of the substring selectivity estimation problem [10, 21, 22]; we refer to [33] for a more extensive description. Compressed storage schemes for such problems help us save space and boost general performance because a larger portion of data can reside in the fast memory. Graph representation of data is gaining importance in the database community. For instance, the set of subject-predicate-object RDF triples can be represented as a graph or as two binary relations [12].

Our compressed representation applied to an RDF graph enables us to support basic reporting and counting queries on triples. An example of such a query is given x, to enumerate all the triples in which x occurs as a subject. Another example is, given x and p, to enumerate all triples in which x occurs as a subject and p occurs as a predicate.

**Previous Results. Static Case.**

We will denote by $|T|$ the number of symbols in a sequence $T$ or in a collection of sequences; $T[i]$ denotes the i-th element in a sequence $T$ and $T[i..j] = T[i]T[i+1]...T[j]$. Suffix trees and suffix arrays are two handbook data structures for the indexing problem. Suffix array keeps (references to) all suffixes $T[i..n]$ in lexicographic order. Using a suffix array, we can find the range of suffixes starting with a query string $P$ in $t_{range} = O(|P| + \log n)$ time; once this range is found, we can locate each occurrence of $P$ in $T$ in $t_{locate} = O(1)$ time. A suffix tree is a compact trie that contains references to all suffixes $T[i..n]$ of a text $T$. Using a suffix trie, we can find the range of suffixes starting with a query string $P$ in $t_{range} = O(|P|)$ time; once this range is found, we can locate every occurrence of $P$ in $T$ in $t_{locate} = O(1)$ time. A large number of compressed indexing data structures are described in the literature; we refer to [30] for a survey. These data structures follow the same two-step procedure for answering a query: first, the range of suffixes that start with $P$ is found in $O(t_{range})$ time, then we locate each occurrence of $P$ in $T$ in $O(t_{locate})$ time. Then we can report all occ occurrences of $P$ in $O(t_{range} + \text{occ} \cdot t_{locate})$ time. We can also extract any substring $T[i..i+\ell]$ of $T$ in $O(t_{extract})$ time. Data structures supporting queries on a text $T$ can be extended to answer queries on a collection $\mathcal{C}$ of texts: it suffices to append a unique symbol $\$i$ at the end of every text $T_i$ from $\mathcal{C}$ and keep the concatenation of all $T_i$ in the data structure.

We list the currently best and selected previous results for static text indexes with asymptotically optimal space usage in Table 1. All listed data structures can achieve different space-time trade-offs that depend on parameter $s$: an index typically needs about $nH_k + o(n \log \sigma) + O(n \log n/s)$ bits and $t_{locate}$ is proportional to $s$. Henceforth $H_k$ denotes the $k$-th order empirical entropy and $\sigma$ denotes the alphabet size$^1$. We assume that $k \leq \log n - 1$ for a constant $0 < \alpha < 1$. $H_k$ is the lower bound on the average space usage of any statistical compression method that encodes each symbol using the context of $k$ previous symbols [27]. The currently fastest such index of Belazzougui and Navarro [6] reports all occurrences of $P$ in $O(|P| + s \cdot \text{occ})$ time and extracts a substring of length $\ell$ in $O(s + \ell)$ time. Thus their query time depends only on the parameter $s$ and the length of $P$. Some recently described indices [3, 6] achieve space usage $nH_k + o(nH_k) + o(n)$ or $nH_k + o(nH_k) + O(n)$ instead of $nH_k + o(n \log \sigma)$.

If we are interested in obtaining faster data structures and can use $O(n \log \sigma)$ bits of space, then better trade-offs between space usage and time are possible [18, 19]. For the sake of space, we describe only one such result. The data structure of Grossi and Vitter [19] uses $O(n \log \sigma)$ bits and reports occurrences of a pattern in $O(|P|/\log \sigma + \log \sigma \cdot \log n + \text{occ} \log^2 n)$ time; see Table 3. We remark that the fastest data structure in Table 1 needs $\Omega(n \log^{1+\alpha} n)$ space to obtain the same time for $t_{locate}$ as in [19]. If a data structure from Table 1 uses $O(n \log \sigma)$ space, then $t_{locate} = \Omega(\log_\sigma n)$.

**Dynamic Document Collections.**

In the dynamic indexing problem, we maintain a collection of documents (strings) under insertions and deletions. An insertion adds a new document to the collection, a deletion removes a document from the collection. For any query substring $P$, we must return all occurrences of $P$ in all documents. When a query is answered, relative positions of occurrences are reported. To be precise, we must report all pairs $(doc, off)$, such that $P$ occurs in a document $doc$ at position $off$. We remark that relative positions of $P$ (with respect to document boundaries) are reported. Hence an insertion or a deletion of a document does not change positions of $P$ in other documents. Indexes for dynamic collections of strings were also studied extensively [9, 25, 8, 23, 26, 15, 24, 16, 20, 32, 31]. The fastest previously known result for the case of large alphabets is described in [31]. Their data structure, that builds on a long line of previous work, uses $nH_k + o(n \log \sigma) + O(n \log n/s) + O(k \log n)$ bits of space, where $\rho$ is the number of documents; queries are answered in $O(|P| \log n \log \log n + \text{occ} \cdot \log \log n \log \log n)$ time and updates are supported in $O(\log n + |T_i| \log n + \log \log n)$ amortized time, where $T_i$ is the document inserted into or deleted from the collection. See Table 2 for some other previous results.

An important component of previous dynamic solutions is a data structure supporting rank and select queries: a sequence $S$ over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ is kept in a data structure so that the i-th occurrence of a symbol $a \in \Sigma$ and the number of times a symbol $a$ occurs in $S[i..i]$ for any $1 \leq i \leq n$ can be computed. Thus progress in dynamic indexing was closely related to progress in dynamic data structures for rank and select queries. In [31] the authors obtain a

---

1Let $S$ be an arbitrary string over an alphabet $\Sigma = \{1, \ldots, \sigma\}$. A context $s_i \in \Sigma^k$ is an arbitrary string of length $k$. Let $n_{s_i,a}$ be the number of times the symbol $a$ is preceded by a context $s_i$ in $S$ and $n_{s_i} = \sum_{a \in \Sigma} n_{s_i,a}$. Then $H_k = -\sum_{s_i \in \Sigma^k} \sum_{a \in \Sigma} n_{s_i,a} \log \frac{n_{s_i,a}}{n_{s_i}}$ is the k-th order empirical entropy of $S$. 

---
Table 1: Asymptotically optimal space data structures for static indexing. Occurrences of a string $P$ can be found in $O(t_{\text{range}} + t_{\text{locate}} \cdot \sigma)$ time. A substring $T[i..i+\ell]$ of $T$ can be extracted in $O(t_{\text{extract}})$ time. Results are valid for any $k \leq \alpha \log n - 1$ and $0 < \alpha < 1$.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Space $\left(+O\left(n \frac{\log^2 n}{\sigma}\right)\right)$</th>
<th>$t_{\text{range}}$</th>
<th>$t_{\text{locate}}$</th>
<th>$t_{\text{extract}}$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[18]</td>
<td>$nH_k + o(n \log \sigma)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma + \log^4 n)$</td>
<td>$O(s \log \sigma)$</td>
</tr>
<tr>
<td>[34]</td>
<td>$nH_k + o(n \log \sigma)$</td>
<td>$O(</td>
<td>P</td>
<td>\log n)$</td>
<td>$O(s \log \sigma)$</td>
</tr>
<tr>
<td>[13]</td>
<td>$nH_k + o(n \log \sigma)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma \log \log n)$</td>
<td>$O(s \log \sigma \log \log n)$</td>
</tr>
<tr>
<td>[5]</td>
<td>$nH_k + o(n \log \sigma)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma \log \log n)$</td>
<td>$O(s \log \sigma \log \log n)$</td>
</tr>
<tr>
<td>[3]</td>
<td>$nH_k + o(n(H_k) + o(n)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma \log \log n)$</td>
<td>$O(s \log \sigma \log \log n)$</td>
</tr>
<tr>
<td>[3]</td>
<td>$nH_k + o(n(H_k) + o(n)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma \log \log n)$</td>
<td>$O(s \log \sigma \log \log n)$</td>
</tr>
<tr>
<td>[6]</td>
<td>$nH_k + o(n(H_k) + o(n)$</td>
<td>$O(</td>
<td>P</td>
<td>\log \sigma \log \log n)$</td>
<td>$O(s \log \sigma \log \log n)$</td>
</tr>
</tbody>
</table>

Our Results.

In this paper we show that the lower bound on dynamic rank-select problem can be circumvented and describe data structures that need significantly less time to answer queries. Our results close or almost close the gap between static and dynamic indexing. If the alphabet size $\sigma = \log^{O(1)} n$, we can obtain an $(nH_k + o(n \log \sigma) + O(n \frac{\log^2 n}{\sigma}))$-bit data structure that answers queries in $O(|P| \log \log n + \sigma \cdot s)$ time; updates are supported in $O(|T_n| \log^4 n)$ time, where $T_n$ denotes the document that is inserted into or deleted from the index. Our second data structure supports updates in $O(|T_n| \log^6 n)$ expected time and answers queries in $O(|P| \log \log n + \sigma \cdot s)$ time for an arbitrarily large alphabet. If the update procedure is deterministic, then queries are answered in $O(|P| \log \log n + \sigma \cdot s)$ time and updates are supported in $O(|T_n| \log^6 n)$ worst-case time. See Table 2. If $O(n \log \sigma)$ bits of space are available, then our dynamic data structure matches the currently fastest static result of Grossi and Vitter [19]. We can report all occurrences of a pattern $P$ in $O(|P| \log\log n + \sigma \cdot n \log \log n)$ time. This is the first compressed dynamic data structure that achieves $t_{\text{range}} = o(|P|)$ if $\sigma = n^{O(1)}$. Compared to the fastest previous data structure that needs the same space, we achieve $O(\log n \log \log n)$ factor improvement in query time. A variant of this data structure with deterministic update procedure answers queries in $O(|P| \log \log n)^3 / \log^2 n + \log \log n + \sigma \cdot n \log \log n)$ time. See Table 3.

Our data structures can also count occurrences of a pattern $P$ in $O(t_{\text{count}})$ time. For previously described indexes $t_{\text{count}} = t_{\text{range}}$. In our case, $t_{\text{count}} = t_{\text{range}} + \log \log n$ or $t_{\text{count}} = (t_{\text{range}} + \log \log n / \log \log n) \log \log n$. Times needed to answer a counting query are listed in Table 4. However, if our data structures support counting queries, then update times grow slightly, as shown in Table 4.

All of the above mentioned results are obtained as corollaries of two general transformations. Using these transformations, that work for a very broad class of indexes, we can immediately turn almost any static data structure with good pre-processing time into an index for a dynamic collection of texts. The query time either remains the same or increases by a very small multiplicative factor. Our method can be applied to other problems where both compressed representation and dynamism are desirable.

2 Dynamic indexes also need $O(\log \log n)$ bits to navigate between documents, where $\rho$ is the number of documents. Since $\rho \log n$ is usually negligible in comparison to $n$, we ignore this additive term, except for Tables 2 and 3, to simplify the description.
Binary Relations and Graphs.

One important area where our techniques can also be used is compact representation of directed graphs and binary relations. Let \( R \subseteq L \times O \) be a binary relation between labels from a set \( L \) and objects from a set \( O \). Barbay et al. [5] describe a compact representation of a static binary relation \( R \) (i.e., the set of object-label pairs) that consists of a sequence \( S_R \) and a bit sequence \( B_R \). \( S_R \) contain the list of labels related to different objects and is ordered by object. That is, \( S_R \) lists all labels related to an object \( o_1 \), then all labels related to an object \( o_2 \), etc. The binary sequence \( B_R \) contains unary-encoded numbers of labels related to objects \( o_1, o_2, \ldots \). Barbay et al [5] showed how \( S_R \) and \( B_R \) can be used to support basic queries on binary relations, such as listing or counting all labels related to an object, listing or counting all objects related to a label, and telling whether a label and an object are related. Their method reduces queries on a binary relation \( R \) to rank, select, and access queries on \( S_R \) and \( B_R \). Another data structure that stores a static binary relation and uses the same technique is described in [2]. Static compact data structures described in [5, 2] support queries in \( O(\log \log \sigma) \) time per reported datum, where \( \sigma \) is the number of labels. For instance, we can report labels related to an object (resp. objects related to a label) in \( O((k+1) \log \log \sigma) \) time, where \( k \) is the number of reported items; we can tell whether an object and a label are related in \( O(\log \log \sigma) \) time. In [31], the authors describe a dynamization of this approach that relies on dynamic data structures answering rank and select queries on dynamic strings \( S_R \) and \( B_R \). Again the lower bound on dynamic rank queries sets the limit on the efficiency of this approach. Since we need \( \Omega(\log n / \log \log n) \) time to answer rank queries, the data structure of Navarro and Nekrich [31] needs \( O(\log n / \log \log n) \) time per reported item, where \( n \) is the number of object-label pairs in \( R \). Updates are supported in \( O(\log n / \log \log n) \) amortized time and the space usage is \( nH + \sigma_1 \log \sigma_1 + t \log t + O(n + \sigma_1 \log \sigma_1) \) where \( n \) is the number of pairs, \( H \) is the zero-order entropy of the string \( S_R \), \( \sigma_1 \) is the number of labels and \( t \) is the number of objects. In [31] the authors also show that we can answer basic adjacency and neighbor queries on a directed graph by regarding a graph as a binary relation between nodes. Again reporting and counting out-going and in-going neighbors of a node can be performed in \( O(\log n) \) time per delivered datum.

In this paper we show how our method for dynamizing compressed data structures can be applied to binary relations and graphs. Our data structure supports reporting labels related to an object or reporting objects related to a label in \( O(\log \log n \cdot \log \log \sigma) \) time per reported datum. We support counting queries in \( O(\log n) \) time and updates in \( O(\log^2 n) \) worst-case time. The same query times are also achieved for the dynamic graph representation. The space usage of our data structures is dominated by \( nH \) where \( n \) is the number of pairs in a binary relation or the number of edges in a graph and \( H \) is the zero-order entropy of the string \( S_R \) defined above. Thus the space usage of our data structure matches that of [31] up to lower-order factors. At the same time we show that reporting queries in a dynamic graph can be supported without dynamic rank and select queries.

Overview.

The main idea of our approach can be described as follows. The input data is distributed among several data structures. We maintain a fraction of the data in an uncompressed data structure that supports both insertions and deletions. We bound the number of elements stored in uncompressed form so that the total space usage of the uncompressed data structure is affordable. Remaining data is kept in several compressed data structures that do not support updates. New elements (respectively new documents) are always inserted into the uncompressed data structure. Deletions from the static data structures are implemented by the lazy deletions mechanism: when a deletion takes place, then the deleted element (respectively the document) is marked as deleted.

We keep positions of marked elements in a data structure, so that all elements in a query range that are not marked as deleted can be reported in \( O(1) \) time per element. When a static data structure contains too much obsolete data (because a certain fraction of its size is marked as deleted), then this data structure is purged: we create a new instance of this data structure that does not contain deleted elements. If the uncompressed data structure becomes too big, we move its content into a (new) compressed data structure. Organization of compressed data structures is inspired by the logarithmic method, introduced by Bentley and Saxe [7]: the size of compressed data structures increases geometrically. We show that re-building procedures can be scheduled in such way that only a small fraction of data is kept in uncompressed form at any given time. Since the bulk of data is kept in static data structures, our approach can be viewed as a general framework that transforms static compressed data structures into dynamic ones.

In Section 2 we describe Transformation 1; Transformation 1, based on the approach outlined above, can be used to turn a static indexing data structure into a data structure for dynamic collection of documents with amortized update cost. The query costs of the obtained dynamic data structure are the same as in the underlying static data structure. In Section 3 we describe Transformation 2 that turns a static indexing data structure into a dynamic data structure with worst-case update costs. We use more sophisticated division into sub-collections and slightly different re-building procedures in Transformation 2. In Section 4 we describe how to obtain new solutions of the dynamic indexing problem using our static-to-dynamic transformations. Finally Section 5 contains our data structures for dynamic graphs and binary relations.

2. Dynamic Document Collections

In this section we show how a static compressed index \( \mathcal{I}_s \) can be transformed into a dynamic index \( \mathcal{I}_d \). \( \mathcal{C} \) will denote a collection of texts \( T_1, \ldots, T_p \). We say that an index \( \mathcal{I}_s \) is \((u(n), w(n))\)-constructible if there is an algorithm that constructs \( \mathcal{I}_s \) in \( O(n \cdot u(n)) \) additional workspace and uses \( O(n \cdot w(n)) \) additional memory. Henceforth we make the following important assumptions about the static index \( \mathcal{I}_s \): \( \mathcal{I}_s \) needs at most \( |S| \phi(S) \) bits of space for any symbol sequence \( S \) and the function \( \phi(\cdot) \) is monotonous: if any sequence \( S \) is a concatenation of \( S_1 \) and \( S_2 \), then \( |S| \phi(S) \geq |S_1| \phi(S_1) + |S_2| \phi(S_2) \).

We also assume that \( \mathcal{I}_s \) reports occurrences of a substring in \( \mathcal{C} \) using the two-step method described in the introduction: first we identify the range \([a, b]\) in the suffix array, such that all suffixes that start with \( P \) are in \([a, b]\); then we find the
Table 3: \(O(n \log \sigma)\)-bit indexes. Dynamic data structures need additional \(\rho \log n\) bits. Randomized update costs are marked with R.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>(t_{\text{range}})</th>
<th>(t_{\text{locate}})</th>
<th>(t_{\text{extract}})</th>
<th>Update</th>
<th>(\sigma)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[19]</td>
<td>(O(</td>
<td>P</td>
<td>/ \log \sigma + \log n))</td>
<td>(O(\log n))</td>
<td>(O(\ell / \log \sigma))</td>
</tr>
<tr>
<td>[8]</td>
<td>(O(</td>
<td>P</td>
<td>\log n))</td>
<td>(O(\log^2 n))</td>
<td>(O(\log n + \ell))</td>
</tr>
<tr>
<td>[31]</td>
<td>(O(</td>
<td>P</td>
<td>\log n))</td>
<td>(O(\log n \log \log n))</td>
<td>(O(\log n + \ell))</td>
</tr>
<tr>
<td>Our</td>
<td>(O(</td>
<td>P</td>
<td>/ \log \sigma + \log n))</td>
<td>(O(\log n))</td>
<td>(O(\ell / \log \sigma))</td>
</tr>
<tr>
<td>Our</td>
<td>(O(</td>
<td>P</td>
<td>/ (\log \sigma)^2 / \log n + \log n))</td>
<td>(O(\log n))</td>
<td>(O(\ell / \log \sigma))</td>
</tr>
</tbody>
</table>

Table 4: Costs of counting queries for our data structures. Randomized update costs are marked with R. The first three rows correspond to the last three rows in Table 2, the last two rows correspond to the last two rows in Table 3.

positions of suffixes from \([a, b]\) in the document(s). These operations will be called range-finding and locating. Moreover the rank of any suffix \(T[i..]\) in the suffix array can be found in time \(O(|I_s|)\). The class of indexes that satisfy these conditions includes all indexes that are based on compressed suffix arrays or the Burrows-Wheeler transform. Thus the best currently known static indexes can be used in Transformation 1 and the following transformations described in this paper.

Our result can be stated as follows.

**Transformation 1.** Suppose that there exists a static \((u(n), w(n))\)-constructible index \(I_s\) that uses \(|S|\phi(S)\) space for any document collection \(S\). Then there exists a dynamic index \(I_d\) that uses \(|S|\phi(S) + O(|S|((\log \sigma + u(n) + \log \sigma)/\log n))\) space for a parameter \(\tau = O(\log n / \log \log n)\); \(I_d\) supports insertions and deletions of documents in \(O(u(n) \log^2 n)\) time per symbol and \(O(u(n) \cdot \tau + \log \sigma + \log^2 n)\) time per symbol respectively. Update times are amortized. The asymptotic costs of range-finding, extracting, and locating are the same in \(I_s\) and \(I_d\).

We start by showing how to turn a static index into a semi-dynamic deletion-only index using \(O(n/\tau) \log \tau\) additional bits. Then we will show how to turn a semi-dynamic index into a fully-dynamic one.

**Supporting Document Deletions.**

We keep a bit array \(B\) whose entries correspond to positions in the suffix array \(SA\) of \(C\). \(B[j] = 0\) if \(SA[j]\) is a suffix of some text \(T_j\), such that \(T_j\) was already deleted from \(C\) and \(B[j] = 1\) otherwise. We keep a data structure \(V\) that supports the following operations on \(B\): \(\text{zero}(j)\) sets the \(j\)-th bit in \(B\) to 0; \(\text{report}(j, j_i)\) reports all 1-bits in \(B[j..j_i]\). \(V\) is implemented using Lemma 3, so that \(\text{zero}(\cdot)\) is supported in \(O(\log n)\) time and \(\text{report}(j, j_i)\) is answered in \(O(k)\) time, where \(k\) is the number of output bit positions. If \(B\) contains at most \(n/\tau\) zeros, then \(B\) and \(V\) need only \(O(n \log \tau)\) bits. Lemma 3 is proved in Section A.1.

When a document \(T_j\) is deleted, we identify the positions of \(T_j\)'s suffixes in \(SA\) and set the corresponding bits in \(B\) to 0. When the number of symbols in deleted documents equals \(n/\tau\), we re-build the index for \(C\) without deleted documents in \(O(n \cdot u(n))\) time. The total amortized cost of deleting a document is \(O(u(n) \tau + t_{\text{SA}} + \log^2 n)\) per symbol. To report occurrences of some string \(P\) in \(C\), we identify the range \([s, e]\) such that all suffixes in \(SA[s..e]\) start with \(P\) in \(O(t_{\text{range}})\) time. Using \(V\), we enumerate all \(j\) such that \(s \leq j \leq e\) and \(B[j] = 1\). For every such \(j\), we compute \(SA[j]\) in \(O(t_{\text{locate}})\) time.

**Fully-Dynamic Index.**

We split \(C\) into a constant number of sub-collections \(C_0, C_1, \ldots, C_r\) such that \(|C_i| \leq \max_i\) for all \(i\). The maximum size of the \(i\)-th sub-collection, \(\max_i\), increases geometrically: \(\max_0 = 2n/\log^2 n\) and \(\max_i = 2(n/\log^2 n) \log^{i+1} n\) for a constant \(\varepsilon > 0\); see Fig. 1. There is no lower bound on the number of symbols in a sub-collection \(C_i\); for instance, any \(C_i\) can be empty. Our main idea is to store \(C_0\) in uncompressed form and \(C_i\) for \(i \geq 1\) in semi-static deletion-only data structures. Insertions into \(C_i\) for \(i \geq 1\) are supported by re-building the semi-static index of \(C_i\). We also re-build all sub-collections when the total number of elements is increased by a constant factor (global re-build).
We store the document collection \( C_0 \) in uncompressed form. Suffixes of all documents in \( C_0 \) are kept in an (uncompressed) suffix tree \( D_0 \). We can insert a new text \( T \) into \( D_0 \) or delete \( T \) from \( D_0 \) in \( O(|T|) \) time. Using \( D_0 \), all occurrences of a pattern \( P \) in \( C_0 \) can be reported in \( O(|P| \cdot \text{occ}) \) time. Since \( |C_0| \leq 2n/\log^2 n \), we need \( O(n/\log n) \) bits to store \( C_0 \). For completeness we describe the data structure for \( C_0 \) in the full version of this paper [29].

Every \( C_i \) for \( i \geq 1 \) is kept in a semi-dynamic data structure described in the first part of this section. Let \( \text{size}(i) \) denote the total length of all undeleted texts in \( C_i \). When a new document \( T \) must be inserted, we find the first (smallest) collection \( C_j \) such that \( \sum_{i=0}^{j-1} \text{size}(i) + |T| \leq \max_j \), where \( \max_j = 2(n/\log^2 n) \log^{j+1} n \). That is, we find the first sub-collection \( C_j \) that can accommodate the new text \( T \) and all preceding sub-collections without exceeding the size limit. If \( j = 0 \), we insert the new text into \( C_0 \). Otherwise, if \( j \geq 1 \), we discard the old indexes for all \( C_i \), where \( 0 \leq i < j \), set \( C_j = (\bigcup_{i=0}^{j-1} C_i) \cup T \) and construct a new semi-static index for \( C_j \). If \( \sum_{i=0}^{j-1} \text{size}(i) + |T| > \max_j \) for all \( j \), we start a global re-build procedure: all undeleted texts from old sub-collections are moved to the new sub-collection \( C_r \), and parameters \( max \) are recalculated; new sub-collections \( C_i \) for \( 0 \leq i < r \) are initially empty after the global re-build.

We start a global re-build procedure when the total number of elements is at least doubled. Hence, the amortized cost of a global re-build is \( O(u(n)) \). The amortized cost of re-building sub-collections can be analyzed as follows. When a sub-collection \( C_j \) is re-built, we insert all symbols from sub-collections \( C_i \), \( 0 \leq i < j \) and the new text \( T \) into \( C_j \). Our insertion procedure guarantees that \( \sum_{i=0}^{j-1} \text{size}(i) + |T| > \max_{j-1} \). We need \( O(\max_j \cdot u(n)) \) time to construct a new index for \( C_j \). The cost of re-building \( C_j \) can be distributed among the new text symbols inserted into \( C_j \). Since \( \max_{j-1} = \max_j / \log^n n \), the amortized cost of inserting a new symbol into \( C_j \) is \( O(u(n)) \cdot \log^n n \). Every text is moved at most once to any sub-collection \( C_j \) for any \( j \) such that \( 1 \leq j \leq \lceil 2/\varepsilon \rceil \). Hence the total amortized cost of an insertion is \( O((1/\varepsilon) u(n)) \cdot \log^n n \) per symbol.

A query is answered by querying all non-empty sub-collections \( C_i \) for \( i = 0, 1, \ldots, r \). Since \( r = O(1) \), query times are the same as in the underlying static index. Splitting a collection into sub-collection does not increase the space usage because the function \( \phi(\cdot) \) is monotonous. We need \( O((n/\tau) \log \tau) \) bits to keep data structures \( V \) for all \( C_i \). Another \( O(nu(n)) \) bits are needed for global and local re-builds. Finally we need \( O((n/\tau) \log \tau) \) bits to store the symbols from deleted documents. Since there are no more than \( O(n/\tau) \) deleted symbols, we use \( O((n/\tau) \log \tau) \) additional bits to store them; a more detailed analysis is given in the full version [29]. Hence, the total space overhead of our dynamic index is \( O(n(w(n)) + (\log \sigma + \log \tau)/\tau)) \).

3. WORST-CASE UPDATES

In this section we will prove the following result.

**Transformation 2.** Suppose that there exists a static \((u(n), w(n))\)-constructible index \( I_\tau \) that uses \(|S|\phi(S) \text{ time for any document collection } S \). Then there exists a dynamic index \( I_\tau \) that uses \(|S|\phi(S) + O(|S| \log^4 n \log \tau + u(n)) \text{ space for any parameter } \tau = (\log n/\log \log n) \text{ space}\) per symbol and \( O(u(n)) \cdot (\log^2 n + \tau \log \tau + \varepsilon) \text{ time per symbol}\) respectively. The asymptotic costs of range-finding increases by \( O(\tau) \); the costs of extracting and locating are the same in \( I_\tau \) and \( I_{\bar{\tau}} \).

We use the index of Transformation 1 as the starting point. First we give an overview of our data structure and show how queries can be answered. Then we describe the procedures for text insertions and deletions.

**Overview.**

The main idea of supporting updates in worst-case is to maintain several copies of the same sub-collection. An old copy of \( C_i \) is locked while a new updated version of \( C_{j+1} \) that includes \( C_j \) is created in the background. When a new version of \( C_{j+1} \) is finished, we discard an old locked sub-collection. When a new document \( T \) must be inserted, we insert it into \( C_0 \) if \(|C_0| + |T| \leq \max_j \). Otherwise we look for the smallest \( j \geq 0 \), such that \( C_{j+1} \) can accommodate both \( C_j \) and \( T \); then we move both \( T \) and all documents from \( C_j \) into \( C_j+1 \). If the new document \( T \) is large, \(|T| > \max_j/2 \), we can afford to re-build \( C_{j+1} \) immediately after the insertion of \( T \). If the size of \( T \) is smaller than \( \max_j/2 \), re-building of \( C_{j+1} \) is postponed. For every following update, we spend \( O(\log n \cdot u(n)) \) time per symbol on creating the new version of \( C_{j+1} \). The old versions of \( C_j, C_{j+1} \) are retained until the new version is completed. If the number of symbols that are marked as deleted in \( C_j \) exceeds \( \max_j/2 \), we employ the same procedure for moving \( C_j \) to \( C_{j+1} \); \( C_j \) is locked and we start the process of constructing a new version \( C_{j+1} \) that contains all undeleted documents from \( C_j \).

The disadvantage of delayed re-building is that we must keep two copies of every document in \( C_j \cup C_{j+1} \) until new \( C_{j+1} \) is completed. In order to reduce the space usage, we keep only a fraction of all documents in sub-collections \( C_i \). All \( C_i \), for \( 0 \leq i \leq r \) will contain \( O(n/\tau) \) symbols, where \( \tau \) is the parameter determining the trade-off between space overhead and query time. The remaining documents are kept in top sub-collections \( T_1, \ldots, T_r \) where \( g > 2\tau \). Top sub-collections are constructed using the same delayed approach. But once \( T_r \) is finished, no new documents are inserted into \( T_r \). We may have to re-build a top collection or merge it with another \( T_j \) when the fraction of deleted symbols exceeds a threshold value \( 1/\tau \). We employ the same re-building-in-the-background approach. However, we will show that the background procedures for maintaining \( T_r \) can be scheduled in such a way that only one \( T_j \) is re-built at any given moment. Hence, the total space overhead due to re-building and storage of deleted elements is bounded by an additive term \( O(n(\log \sigma + w(n))/\tau)) \).

**Data Structures.**

We split a document collection \( C \) into sub-collections \( C_0, C_1, \ldots, C_{r-1} \) and top sub-collections \( T_1, \ldots, T_r \) where \( g = O(\tau) \). We will also use auxiliary collections \( N_1, \ldots, N_{r+1} \) and temporary collections \( Temp_1, \ldots, Temp_r \). \( Temp_r \) are also used to answer queries but each non-empty
Temp_i contains exactly one document; Temp_i are used as temporary storage for new document that are not yet inserted into “big” collections. The sizes of sub-collections can be defined as a function of parameter n_f such that n_f = Θ(n); the value of n_f changes when n becomes too large or too small. Let max_i = \(2(n_f/\log^r n) \log^{r-1} n\). We maintain the invariant |C_i| ≤ max_i for all i, 0 ≤ i ≤ r, but r is chosen in such way that n_f/\(\log^r n\)· n_{f,T} = n_f/\(\sqrt{r}\). Every Timelapse contains \(\Omega(n_f/\tau)\) symbols. If Timelapse contains more than one text, then its size is at most 4n_f/\(\sqrt{r}\); otherwise Timelapse can be arbitrarily large. When a collection C_i is locked in the process of re-building C_i can be distributed among a number of future updates (insertions and deletions of documents).

During this time C_i is locked: we set C_i = C_i and initialize a new empty sub-collection C_i. When a new sub-collection \(N_{i+1} = N_i + C_i\) is completed, we set \(N_{i+1} = N_{i+1} + C_i\) and discard old C_i and C_i. A query is answered by querying all non-empty C_i, L_i, Temp_i, and Timelapse. Therefore the cost of answering a range-finding query grows by O(\(\tau\)). The costs of locating and extracting are the same as in the static index. We show main sub-collections used by our method on Figure 2.

Insertions.

When a document T is inserted, we consider all j, such that 0 ≤ j ≤ r and the data structure L_j is not empty. For every such j, we spend O(\(|T| \log \cdot n \cdot u(n)\)) units of time on constructing \(N_{j+1}\). If \(N_{j+1}\) for some 0 ≤ j ≤ r - 1 is completed, we set \(C_{j+1} = N_{j+1}\) and \(N_{j+1} = Temp_{j+1} = 0\); if \(N_{r+1}\) is completed, we set \(T_{r+1} = L_{r+1}\) and increment the number of top collections g, and set \(T_{r+1} = Temp_{r+1} = 0\). Then we look for a sub-collection that can accommodate the new document T. If \(|T| ≥ n/\tau\), we create the index for a new sub-collection \(T_i\) that contains a single document T. If \(|T| < n/\tau\), we look for the smallest index j, such that \(|C_{j+1}| + |C_j| + |T| ≤ \text{max}_i\). That is, \(C_{j+1}\) can accommodate both the preceding sub-collection C_i and T. If \(|T| ≥ \text{max}_i/2\), we set \(C_{j+1} = C_j ∪ C_{j+1} ∪ T\) and create an index for the new \(C_{j+1}\) in O(|\(C_{j+1}\)| · u(n)) = O(|T| \log^r n · u(n)) time. If \(|T| < \text{max}_i/2\), the collection C_j is locked. We set \(T_j = C_j\), \(C_j = 0\) and initiate the process of creating \(N_{j+1} = C_j ∪ C_{j+1} ∪ T\). The cost of creating the new index for \(N_{j+1}\) will be distributed among the next max_j update operations. We also create a temporary static index \(Temp_{j+1}\) for the text T in O(|T|\(u(n)\)) time. This procedure is illustrated on Fig. 3. If the index j is not found and \(|C_i| + |C_{i+1}| + |T| > \text{max}_i\) for all i, 0 ≤ i < r, we lock \(T_j\) (that is, set \(L_j = C_j\) and \(C_j = 0\)) and initiate the process of constructing \(N_{r+1} = L_r ∪ T\). We also create a temporary index \(Temp_{r+1}\) for the document T in O(|T|\(u(n)\)) time.

Deletions.

Indexes for sub-collections C_i, 1 ≤ i ≤ r, and \(T_j\), 1 ≤ j ≤ g, support lazy deletions in the same way as in Section 2: when a document is deleted from a sub-collection, we simply mark the positions of suffixes in the suffix array as deleted and set the corresponding bits in the bit vector B to 0. Augmenting an index so that lazy deletions are supported is done in exactly the same way as in Section 2.

We will need one additional sub-collection \(C'_j\) to support deletions. If a sub-collection C_i for 1 ≤ j ≤ r - 1 contains \(\text{max}_j/2\) deleted elements, we start the process of re-building C_i and merging it with \(C_{j+1}\). This procedure is the same as in the case of insertions. We lock \(C_j\) by setting \(L_j = C_j\) and \(C_j = 0\). The data structure \(N_{j+1} = C_j ∪ L_j\) will be re-built during the following max_j/2 updates. If a sub-collection C_i contains \(\text{max}_j/2\) deleted symbols, we set \(C'_i = C_i\), \(C_i = 0\). The sub-collection \(C'_i\) will be merged with the next sub-collection \(T_i\) to be re-built.

If a collection \(T_i\) contains a single document and this document is deleted, then \(T_i\) is discarded. We also bound the number of deleted symbols in any \(T_i\) by \(n_f/\tau\). This is achieved by running the following background process: After each series of \(n_f/(2\tau \log \tau)\) symbol deletions, we identify \(T_j\) that contains the largest number of deleted symbols. During the next \(n_f/(2\tau \log \tau)\) symbol deletions we build the new index for \(T_j\) without the deleted symbols. At the same time we remove the deleted symbols from \(C'_j\), if \(C'_j\) exists. If \(C'_j\) exists and contains at least \(n_f/2\) undeleted symbols, we create an index for a new sub-collection \(T'_{j+1}\) and increment the number of top collections by g. If \(C'_j\) exists, but contains less than \(n_f/2\) undeleted symbols, we merge \(C'_j\) with the largest \(T_j\) that contains more than one document and split the result if necessary: if the number of undeleted symbols in \(C'_j ∪ T_j\) does not exceed \(n_f/(2\tau \log \tau)\).
not exceed $2n_f/\tau$, we construct an index for $T_j \cup L_i'$ without deleted symbols; otherwise, we split $T_j \cup L_i'$ into two parts $T_j^1$, $T_j^2$ and create indexes for the new sub-collections. Our method guarantees that the number of deleted elements in any collection $T_i$ does not exceed $O(n_f/\tau)$ as follows from a Theorem of Dietz and Sleator [11].

**Lemma 1** ([11], Theorem 5). Suppose that $x_1, \ldots, x_g$ are variables that are initially zero. Suppose that the following two steps are iterated: (i) we add a non-negative real value $a_i$ to each $x_i$ such that $\sum a_i = 1$ (ii) set the largest $x_i$ to 0. Then at any time $x_i \leq 1 + h_i - 1$ for all $i$, $1 \leq i \leq g$, where $h_i$ denotes the $i$-th harmonic number.

Let $m_i$ be the number of deleted elements in the $i$-th top collection $T_i$ and $\delta = n_f/(2\tau \log \tau)$. We define $x_i = m_i/\delta$. We consider the working of our algorithm during the period when the value of $n_f$ is fixed. Hence, $\delta$ is also fixed and the number of variables $x_i$ is $O(\tau)$ (some $x_i$ can correspond to empty collections). Every iteration of the background process sets the largest $x_i$ to 0. During each iteration $\sum x_i$ increases by 1. Hence, the values of $x_i$ can be bounded from above by the result of Lemma 1: $x_i \leq 1 + h_i - 1$ for all $i$ at all times. Hence $m_i = O((n_f/\tau \log \tau)/\log \tau) = O(n_f/\tau)$ for all $i$ because $h_i = O(\log i)$. Thus the fraction of deleted symbols in each $T_i$ is $O(1/\tau)$.

![Figure 3: Suppose that $C_{j+1}$ is the first sub-collection that can accommodate both $C_j$ and a new document $T_n$. If $C_j$ must be rebuilt in the background, we "rename" $C_j$ to $L_j$ and initialize another (initially empty) $C_j$. New document $T_n$ is put into a separate collection $Temp_{j+1}$ (a). A background process creates a new collection $N_{j+1}$ that contains all documents from $L_j$, $C_{j+1}$ and $Temp_{j+1}$ (b). When $N_{j+1}$ is finished, we discard $C_{j+1}$, $L_j$ and $Temp_{j+1}$, and set $C_{j+1} \equiv N_{j+1}$ (c). Our procedure guarantees that $N_{j+1}$ is completed before the new sub-collection $C_j$ must be re-built again.](image)

It is easy to show that the sub-collections that we use are sufficient for our algorithm. When a sub-collection $L_j$ is initialized, $C_j$ is empty. The situation when $C_j$ cannot accommodate a new document $T_n$ and a preceding sub-collection $C_{j-1}$ can happen only after $max_j - \sum_{i=1}^{j-1} max_i$ new symbol insertions. Since we spend $O(\log n \cdot u(n))$ time for constructing $N_{j+1}$ with each new symbol insertion, we can choose constants in such a way that construction of $N_{j+1}$ is finished (and $L_j$ is discarded) after $max_j/2 < max_j - \sum_{i=1}^{j-1} max_i$ symbol insertions. The situation when $C_j$ contains $max_j/2$ deleted symbols cannot happen after at least $max_j$ new symbol updates (max $j/2$ insertions and max $j/2$ deletions). Hence, the collection $L_j$ is discarded before $C_j$ has to be locked again. In our description of update procedures we assumed that the parameter $n_f$ is fixed. We can maintain the invariant $n_f = \Theta(n)$ using standard methods; for completeness we will provide a description in the full version [29].

The space overhead caused by storing copies of deleted elements is bounded by $O(n/\tau)$; all $C_j$ contain $O(n/\tau)$ symbols and at most every second symbol in each $C_j$ is from a deleted document; the fraction of deleted symbols in each $C_j$ does not exceed $O(1/\tau)$. By the same argument, at any moment of time at most $O(n/\tau)$ symbols are in sub-collections that are re-built. Hence re-building procedures running in the background need $O(nu(n)/\tau)$ bits of space. Since each $T_i$ contains at most $O(|T_i|/\tau)$ deleted symbols, we can store the data structure $V$, which enables us to identify undeleted elements in any range of the suffix array and is implemented as described in Lemma 3, using $O(|T_i| \log \tau/\tau)$ bits. Data structures $V$ for all $T_i$ need $O(n \log \tau/\tau)$ bits. Hence, the total space overhead of $\mathcal{I}_d$ compared to $\mathcal{I}_a$ is $O((n + \log \tau + \log \tau)/\tau)$ bits.

**Counting Occurrences.**

Our dynamic indexes can be easily extended so that pattern counting queries are supported.

**Theorem 1.** We can augment the indexes $\mathcal{I}_d$ of Transformations 1-2 with $O(n \log \tau/\tau)$ additional bits so that all occurrences of a pattern can be counted in $O(\text{count})$ time, where $\text{count} = (\text{range} + \log n / \log \log n)(\tau + \tau)$ and $\tau$ is defined as in the proofs of respective Transformations. If counting is supported, update times are increased by $O(\log n / \log \log n)$ additive term per symbol.

**Proof.** Every semi-dynamic index for a sub-collection $C_i$ (respectively $T_i$) already keeps a vector $B$ that enables us to identify the suffixes of already deleted documents in the suffix array. We also store each $B$ in a data structure of Navarro and Sadakane [32] that supports rank queries in $O(\log n / \log \log n)$ time and updates in $O(\log n / \log \log n)$ time. If $B$ contains $O(B/\tau)$ zero values, then the structure of [32] needs $O(|B|/\tau) \log \tau$ bits. Using this data structure, we can count the number of 1’s in any portion $B[a..b]$ of $B$ in the same time. To answer a counting query, we first answer a range-finding query in every sub-collection. For every non-empty range that we found, we count the number of 1’s in that range. Finally, we sum the answers for all sub-collections. Since a range-finding query returns the range of all suffixes that start with a query pattern and each
1 in $V$ corresponds to a suffix of an undeleted document, our procedure is correct. □

4. DYNAMIC INDEXES

To obtain our results on dynamic document collections, we only need to plug some currently known static indexes into Transformations 1 and 2. We prove the statements about constructibility of static indexes in the full version of this paper [29].

The static index of Belazzougui and Navarro, described in [6], is $(log \ n, log \ σ)$-constructible. Their index achieves $t_{range} = O(|P|), t_{extract} = O(s + ℓ), t_{ISA} = O(s)$ for arbitrarily large alphabets; it needs $nH_k + O(\frac{n log n}{δ}) + O(n log log n) + O(n)$ bits. We apply Transformation 2 with $τ = log log n$. The construction algorithm for this relies on randomized algorithm for constructing an mmph functions [6]; therefore the update procedures of our dynamic data structure also rely on randomization in this case. The resulting dynamic index uses $nH_k + O(\frac{n log n}{δ}) + O(n log log n) + O(n)$ bits. This index achieves $t_{range} = O(|P| log log n), t_{extract} = O(s + ℓ), t_{locate} = O(s)$. Inserts and deletions are supported in $O(|T| log^2 n)$ time and $O(|T| log^2 n + s)$ expected time respectively. If counting queries are also supported, then $t_{count} = O(|P| log log n + log n)$ and updates take $O(|T| log log n)$ expected time.

The index of Barbay et al. [3, 2] is also $(log \ n, log \ σ)$-constructible and uses $nH_k + O(\frac{n log n}{δ}) + O(n log log n)$ time. If the alphabet size $σ = 2^{O(1)}$, this index achieves $t_{range} = O(|P|), t_{extract} = O(s + ℓ), and t_{locate} = O(s)$; it uses $nH_k + O(\frac{n log n}{δ}) + O(n log log n)$ bits. If we set $τ = log log n$ and apply Transformation 2, we obtain a dynamic data structure with $t_{range} = O(|P| log log n), t_{extract} = O(s + ℓ), and t_{locate} = O(s)$. An arbitrary alphabet size $σ$, the index of Barbay et al. [3, 2] achieves $t_{range} = O(|P| log log σ), t_{extract} = O(s + ℓ log log σ), and t_{ISA} = O(s log log σ). Again we set $τ = log log n$ and apply Transformation 2. We obtain a dynamic index that has query costs $t_{range} = O(|P| log log σ log σ log n), t_{extract} = O((s + ℓ) log log σ), and $t_{ISA} = O(s log log σ). Inserts and deletions are supported in $O(|T| log^2 n)$ time and $O(|T| log^2 n + s)$ time respectively. If counting queries are also supported, then $t_{count} = O(|P| log log n log σ + log n)$ (resp. $t_{count} = O(|P| log log n + log n)$ if $σ = log^{O(1)} n$) and updates take $O(|T| log log n)$ time.

The index of Grossi and Vitter [19] is $(log^ε n, log \ σ)$-constructible. It achieves $t_{locate} = O(log^ε n), t_{range} = O(|P| log log n + log^ε n)$ and $t_{extract} = O(ℓ / log_σ n)$. We apply Transformation 2 with $τ = 1/δ$ for a constant $δ$. The resulting dynamic index uses $O(n log σ(1 + 1/δ)) = O(n log σ)$ bits and has the following query costs: $t_{locate} = O(log^2 n), t_{range} = O(|P| log log n + log^ε n), t_{extract} = O(ℓ / log_σ n)$. As shown in the full version [29], in this case the data structure for uncompressed sequence $G_0$ relies on hashing. Therefore the update procedure is randomized. Updates are supported in $O(|T| log^2 n)$ expected time, but we can replace $ε$ with $ε/2$ in our construction and reduce the update time to $O(|T| log^ε n)$. If counting queries are also supported, then $t_{count} = O(|P| log log n + log log log n)$ and updates take $O(|T| log log n)$ expected time. If we want to support updates using a deterministic procedure, then the cost of searching in $C_0$ grows to $O(|P| log log n)^2 / log_σ n + log n)$. In this case $t_{range} = t_{count} = O(|P| log log n|^2 / log_σ n + log n)$, $t_{locate} = O(log^ε n)$, and $t_{extract} = O(ℓ / log_σ n)$.

5. DYNAMIC GRAPHS AND BINARY RELATIONS

Let $R$ denote a binary relation between $t$ objects and $σ_t$ labels. In this section we denote by $n$ the cardinality of $R$, i.e., the number of object-label pairs. We will assume that objects and labels are integers from intervals $[1, σ_t]$ and $[1, t]$ respectively. Barbay et al. [4] showed how a static relation $R$ can be represented by a string $S$. A dynamization of their approach based on dynamic data structures for rank and select queries is described in [31].

Let $M$ be a matrix that represents a binary relation $R$: columns of $R$ correspond to objects and rows correspond to matrices. The string $S$ is obtained by traversing $M$ column-wise (i.e., objectwise) and writing the labels. An additional bit string $N$ encodes the numbers of labels related to objects: $N = 1^n 01^n 0 \ldots \ldots 1^n$, where $n_l$ is the number of labels related to the $l$-th object. Using rank, select, and access queries on $N$ and $S$, we can enumerate objects related to a label, enumerate labels related to an object, and decide whether an object and a label are related.

Deletion-Only Data Structure.

We keep $R$ in $S$ and $N$ described above; $S$ and $N$ are stored in static data structures. If a pair $(e, l)$ is deleted from $R$, we find the element of $S$ that encodes this pair and mark it as deleted. We record marked elements (i.e. pairs that are deleted but are still stored in the data structure) in a bit vector $D$: $D[i] = 0$ if and only if the pair $(S[i], l)$ is marked as deleted. We maintain the data structure of Lemma 3 on $D$. Moreover we keep $D$ in a data structure described in [17]; this data structure enables us to count the number of 1-bits in any range of $D$. For each label $a$ we also keep a data structure $D_a$. $D_a$ is obtained by traversing the $a$-th row of $M$: if $M[a, j] \neq 0$, then we append 0 to $D_a$ if $(a, j)$ is marked as deleted; if $M[a, j] = 0$ and $(a, j)$ is not marked as deleted, we append 1 to $D_a$. For each $D_a$ we also maintain data structures for reporting and counting 1-bits described above. Finally we record indices of deleted labels and objects in two further bit sequences. The static data structures on $S$ and $N$ are implemented as in [29], so that rank and select queries are answered in $O(log log σ_t)$ time and any $S[i]$ or $N[i]$ can be retrieved in constant time.

If we need to list labels related to an object $i$, we first find the part of $S$ that contains these labels. Let $l = rank_k(\text{select}_0(i - 1, N), N)$ and $r = rank_k(\text{select}_0(i, N), N)$. We list all elements of $S[l..r]$ that are not marked as deleted by enumerating all 1-bits in $D[l..r]$. Then we access and report $S[i_1], S[i_2], \ldots, S[i_l]$, where $i_1, i_2, \ldots, i_l$ are positions of 1-bits in $D[l..r]$. In order to list objects related to a label $σ$, we find positions of 1-bits in $D_a$. Then we access and report $\text{select}_0(j_1, S), \text{select}_0(j_2, S), \ldots, \text{select}_0(j_l, S)$. In order to determine whether an object $i$ and a label $a$ are related, we compute $d = rank_k(r, S) - rank_k(l, S)$, where $l$ and $r$ are as defined above. If $d = 0$, then the object $i$ and the label $a$ are not related. If $d = 1$, we compute $j = \text{select}_0(rank_k(r, S), i)$ and $i$ and $a$ are related if and only if $D[j] = 1$.

When $(e, l)$ is deleted, we find the position $j$ of $(e, l)$ in $S$ and set $D[j] = 0$; $j$ can be found with a constant num-
ber of rank and select queries. We also set \( D_0[j'] = 0 \) for \( j' = \text{rank}_0(S, j) \). When an empty label or an empty object is removed, we simply record this fact by adding it to a compact list of empty labels (resp. empty objects). When the number of pairs that are marked as deleted exceeds \( n/\tau \), we start the process of re-building the data structure. The cost of re-building is distributed among the following updates; we will give a more detailed description in the exposition of the fully-dynamic data structure.

**Fully-Dynamic Data Structure.**

We split a binary relation \( R \), regarded as a set of object-label pairs, into subsets and keep these subsets in data structures \( C_0, C_1, \ldots, C_r, L_1, \ldots, L_r, T_1, \ldots, T_r \) for \( g = \Theta(\tau) \). We set the parameter \( \tau = \log \log n \). Only \( C_0 \) is stored in a fully-dynamic data structure, but we can afford to keep \( C_0, C_1, \ldots, C_r \) per item because it contains only a small fraction of pairs. All other pairs are stored in deletion-only data structures described above. Distribution of pairs among subsets and procedures for re-building deletion-only data structures are the same as in Section 3. To simplify a description, we will not distinguish between a subset and a data structure that stores it.

\( C_0 \) contains at most \( \max x = 2n/\log^2 n \) pairs. Each structure \( C_i \) for \( r \geq 1 \) contains at most \( \max x = 2n/\log^{2-\varepsilon} n \) pairs. Every \( T_i \) contains at most \( 2n/\tau \) pairs. Data structure \( C_0 \) contains object-label pairs in uncompressed form and uses \( O(\log n) \) bits per pair. For every object \( i \) that occurs in \( C_0 \) we keep a list \( L_i \) that contains all labels that occur in pairs \( (i, \cdot) \in C_0 \); for each label \( a \) that occurs in \( C_0 \) we keep a list of objects that occur in pairs \( (\cdot, a) \in C_0 \). Using these lists we can enumerate all objects related to a label or labels related to an object in \( C_0 \) in \( O(1) \) time per datum. If we augment lists \( L_i \) with predecessor data structures described in [1], we can also find out whether an object \( i \) and a label \( a \) are related in \( O((\log \log n)^2) \) time.

All pairs in \( C_1, L_1, \ldots, C_r, L_r, \) and \( T_1, T_r \) are kept in deletion-only data structures described above. A new object-label pair \( (i, a) \) is inserted into \( C_0 \) if \( C_0 \) contains less than \( \max x \) pairs. Otherwise we look for the smallest \( j \) with \( 0 \leq j < r \), such that \( |C_j| + |C_{j+1}| + 1 \leq \max x + 1 \). We lock \( C_j \) by setting \( L_j = C_j \), and then we initiate the process of creating \( N_j = C_j \cup \bigcup_{0 \leq i < j} \{(i, a)\} \). If \( |C_j| + |C_{j+1}| + 1 \leq \max x + 1 \) for all \( i < r \), we lock \( C_r \) and start the process of constructing \( N_{r+1} = C_r \cup \{(i, a)\} \). The cost of creating \( N_j \) is distributed among the next max updates, in the same way as in Section 3. We observe that data structures \( Temp_j \) are not needed now because each update inserts only one element (pair) into the relation \( R \). We guarantee that each structure \( C_i \) for some \( 1 \leq i \leq r \) contains at most \( \max x/2 \) pairs marked as deleted and \( T_{i+1} \) for \( 1 \leq i \leq r \) contains an \( O(1/\tau) \) fraction of deleted pairs. Procedures for re-building data structures that contain too many pairs marked as deleted are the same as in Section 3.

Our fully-dynamic data structure must support insertions and deletions of new objects and labels. An object that is not related to any label or a label that is not related to any object can be removed from a data structure. This means that both the number of labels \( \sigma_i \) and the number of objects \( t \) can change dynamically. Removing and inserting labels implies changing the alphabets of strings \( S \) that are used in deletion-only data structures. Following [31] we store two global tables, \( NS \) and \( SN \); \( SN \) maps labels to integers bounded by \( O(\sigma_i) \) (global label alphabet) and \( NS \) maps integers back to labels. We also keep bitmaps \( GC_i \) and \( GT_i \), \( GL_i \), and \( GN_i \) for all subsets \( C_i, L_i, N_i, \) and \( T_i \). \( GC_i[j] = 1 \) if the label that is assigned to integer \( j \) occurs in \( C_i \) and \( GC_i[j] = 0 \) otherwise; \( GT_i \), \( GL_i \), and \( GN_i \) keep the same information for subsets \( T_i, L_i, \) and \( N_i \). Using these bit sequences we can map the symbol of a label in the global alphabet to the symbol of the same label in the effective alphabet \(^4\) used in one of subsets. When a label \( a \) is deleted, we mark \( SN[a] \) as free. When a new label \( a' \) is inserted, we set \( SN[a'] \) to a free slot in \( SN \) (a list of free slots is maintained). When some subset, say \( C_i \), is re-built, we also re-build the bit sequence \( GC_i \).

In order to list objects related to a label \( a \), we first report all objects that are related to \( SN[a] \) and stored in \( C_0 \). Then we visit all subsets \( C_i, L_i, \) and \( T_i \), and report all objects related to \( rank_1(SN[a], GC_i), rank_k(SN[a], GL_i), \) and \( rank_k(SN[a], GT_i) \) respectively. We remark that a global symbol of a label can be mapped to a wrong symbol in the local effective alphabet. This can happen if some label \( a' \) is removed and its slot in \( SN[\cdot] \) is assigned to another label \( a \) but the bitmap of say \( GC_i \) is not yet re-built. In this case \( rank_k(SN[a], GC_i) \) will map to the symbol for the wrong label \( a' \). But \( a' \) can be removed only if all object-label pairs containing \( a' \) are deleted; hence, all pairs \( (i, a') \) in \( C_0 \) are marked as deleted and the query to \( C_i \) will correctly report nothing. We can report labels related to an object and tell whether a certain object is related to a certain label using a similar procedure. We visit \( O(\log \log n) \) data structures in order to answer a query. In all data structures except for \( C_0 \), we spend \( O(\log \log n) \) time per reported datum. An existential query on \( C_0 \) takes \( O((\log \log n)^2) \) time; all other queries on \( C_0 \) take \( O(1) \) time per reported datum. Hence all queries are answered in \( O(\log \log n \log \log n) \) time per reported datum. A counting query takes \( O(\log n / \log \log n) \) time in each subset. Hence, we can count objects related to a label or labels related to an object in \( O(\log n) \) time.

All bit sequences \( D_0 \) and \( D_d \) in all subsets use \( O((n/\tau) \log \tau) \) bits. Every string \( S \) stored in a deletion-only data structure needs \( |S| H_2(S) + O(|S| \log \sigma_i) \) bits. Hence all strings \( S \) use at most \( nH + o(n \log \sigma_i) \) bits, where \( H = \sum_{0 \leq i \leq \log n} \frac{n}{2^i} \log 2^i \). Bit sequences \( GC_i, GL_i, \) and \( GT_i \) use \( O(\sigma_i \log \log n) \) bits. Now we consider the space usage of bit sequences \( N \) stored in deletion-only data structures. Let \( m_i \) denote the number of pairs in a data structure \( T_i, N \) consists of \( m_i \) 1’s and \( t \) 0’s. If \( m_i > t \), then the bit sequence \( N \) stored as a part of \( T_i \) uses \( m_i \log \left( \frac{m_i + 1}{m_i} \right) = O(m_i) \) bits. If \( t \geq m_i \), \( N \) uses \( O(m_i \log \tau) \) bits because \( m_i = \Theta(n/\tau) \). Hence all \( N \) stored in all \( T_i \), use \( O(n \log \tau) \) bits. In our data structure we set \( \tau = \log \log n \). If \( \sigma_i = \Omega(\log^{1/4} n) \), \( O(n \log \tau) = o(n \log \sigma_i) \). Otherwise \( t = \Omega(n/\log n) \) because \( n \leq t \cdot \sigma_i \); if \( t = \Omega(n/\log n) \), \( O(n \log \tau) = o(\log t) \). Data structures that are re-built at any moment of time contain \( O(n/\tau) \) elements and use \( O(\frac{n}{\tau} \log \log n) = O(n \log \sigma_i) \) bits. Extra space that we need to store elements marked as deleted is bounded by \( o(n \log \sigma_i) \); this can be shown in the same way as in Section 3.

**Theorem 2.** A dynamic binary relation that consists of \( n \) pairs relating \( t \) objects to \( \sigma_i \) labels can be stored in \( nH + o(n \log \sigma_i) + O(t \log t) + O(t + n + \sigma_i \log n) \) bits where \( H = \)

\(^4\)An effective alphabet of a sequence \( S \) contains only symbols that occur in \( S \) at least once.
1. \sum_{1 \leq a \leq \sigma_i} \frac{n_a}{\sigma_i} \log \frac{\sigma_i}{n_a} \text{ and } n_a \text{ is the number of objects related to a label } a. \text{ We can determine whether an object and a label are related in } O(\log \log \sigma_i \log \log n) \text{ time and report all objects related to a label (resp. all labels related to an object) in } O((k + 1) \log \log \sigma_i \log \log n) \text{ time, where } k \text{ is the number of reported items. We can count objects related to a label or labels related to an object in } O(\log n) \text{ time. Updates are supported in } O(\log^a n) \text{ time.}

Directed graph is a frequently studied instance of a binary relation. In this case both the set of labels and the set of objects are identical with the set of graph nodes. There is an edge from a node \( u \) to a node \( v \) if the object \( u \) is related to the label \( v \).

**Theorem 3.** A dynamic directed graph that consists of \( \sigma_i \) nodes and \( n \geq \sigma_i \) edges can be stored in \( nH + o(n \log \sigma_i) + O(n + \sigma_i \log n) \) bits where \( H = \sum_{1 \leq a \leq \sigma_i} \frac{n_a}{\sigma_i} \log \frac{\sigma_i}{n_a} \) and \( n_a \) is the number of outgoing edges from node \( a \). We can determine if there is an edge from one node to another one in \( O(\log \log \sigma_i \log \log n) \) time and report all neighbors (resp. reverse neighbors) of a node in \( O((k + 1) \log \log \sigma_i \log \log n) \) time, where \( k \) is the number of reported nodes. We can count neighbors or reverse neighbors of a node in \( O(\log n) \) time. Updates are supported in \( O(\log^a n) \) time.

6. CONCLUSIONS

In this paper we described a general framework for transforming static compressed indexes into dynamic ones. We showed that, using our framework, we can achieve the same or almost the same space and time complexity for dynamic indexes as was previously obtained by static indexes. Our framework is applicable to a broad range of static indexes that includes a vast majority of currently known results in this area. Thus, using our techniques, we can easily modify almost any compressed static index, so that insertions and deletions of documents are supported. It will likely be possible to apply our framework to static indexes that will be obtained in the future. Our approach also significantly reduces the cost of basic queries in compact representations of dynamic graphs and binary relations. We expect that our ideas can be applied to the design of other compressed data structures.

**Acknowledgments.**

The authors wish to thank Djamal Belazzougui for clarifying the construction time of the static index in [6] and Gonzalo Navarro for explaining some technical details of dynamic indexes used in [26].

7. REFERENCES

APPENDIX

A.1. REPORTING 1-BITS IN A BIT VECTOR

We show how to store a bit vector with a small number of zeros in small space, so that all 1-values in an arbitrary range can be reported in optimal time. This result is used by our method that transforms a static index into an index that supports deletions. We start by describing an $O(n)$-bit data structure. Then we show how space usage can be reduced to $O((n \log \tau)/\tau)$

Lemma 2. There exists an $O(n)$-bit data structure that supports the following operations on a bit vector $B$ of size $n$: (i) zero(i) sets $B[i] = 0$ (ii) report(s,e) enumerates all $j$ such that $s \leq j \leq e$ and $B[j] = 1$. Operation zero(i) is implemented in $O(\log^* n)$ time and a query report(s,e) is answered in $O(k)$ time, where $k$ is the number of output bit positions.

Proof. We divide the vector $B$ into words $W_1, \ldots, W_{\lceil \log n \rceil}$ of $\log n$ bits. We say that a word $W_i$ is non-empty if at least one bit in $W_i$ is set to 1. We store the indices of all non-empty words in a data structure that supports range reporting queries in $O(k)$ time, where $k$ is the number of reported elements, and updates in $O(\log^* n)$ time [28]. For every word $W_i$ we can find the rightmost bit set to 1 before the given position $p$ or determine that there is no bit set to 1 to the right of $p$ in $O(1)$ time. This can be done by consulting a universal look-up table of size $o(n)$ bits. To report positions of all 1-bits in $B[s..e]$, we find all non-empty words whose indices are in the range $[s/\log n], [e/\log n]$. For every such word, we output the positions of all 1-bits. Finally, we also examine the words $W_{\lceil s/\log n \rceil}$ and $W_{\lceil e/\log n \rceil}$ and report positions of 1-bits in these two words that are in $B[s..e]$. The total query time is $O(k)$. Operation zero(i) is implemented by setting the bit $i - \lfloor i/\log n \rfloor \log n$ in the word $W_{\lceil i/\log n \rceil}$ to 0. If $W_{\lceil i/\log n \rceil}$ becomes empty, we remove $\lfloor i/\log n \rfloor$ from the range reporting data structure.

Lemma 3. Let $B$ be a bit vector of size $n$ with at most $O(\frac{n}{\tau})$ zero values for $\tau = O(\log n/\log \log n)$. $B$ can be stored in $O(n \log \tau)$-bit data structure that supports the following operations on $B$: (i) zero(i) sets $B[i] = 0$ (ii) report(s,e) enumerates all $j$ such that $s \leq j \leq e$ and $B[j] = 1$. Operation zero(i) is supported in $O(\log^* n)$ time and a query report(s,e) is answered in $O(k)$ time, where $k$ is the number of output bit positions.

Proof. We divide $B$ into words $W_i$ of $\tau$ bits. Indices of non-empty words are stored in the the data structure $B'$, implemented as in Lemma 2. Every word $W_i$ is represented as follows: we store the number of zeros in $W_i$ using $O(\log \tau)$ bits. A subword with $f$ zeros, where $0 \leq f \leq \tau$, is encoded using $f(\log \tau)$ bits by specifying positions of 0-bits. For every word $W_i$ we can find the rightmost bit set to 1 before the given position $p$ or determine that there is no bit set to 1 to the right of $p$ in $O(1)$ time. This can be done by consulting a universal look-up table of size $o(n)$ bits. Query processing is very similar to Lemma 2. To report 1-bits in $B[s..e]$, we find all non-empty words whose indices are in the range $[s/\tau], [e/\tau]$ for every such word, we output the positions of all 1-bits. Finally, we also examine the words $W_{\lceil s/\tau \rceil}$ and $W_{\lceil e/\tau \rceil}$ and report positions of 1-bits in these two words that are in $B[s..e]$.
Operation \textit{zero}(i) is implemented by setting the corresponding bit in some word \( W_i \) to 0 and changing the word encoding. If \( W_i \) becomes empty, the \( i \)-th bit in \( B' \) is set to 0. Issues related to memory management can be resolved as in [31].

We need \( O(n/\tau) \) bits to store the data structure \( B' \) for non-empty words. Let \( n_f \) denote the number of words with \( f \) zero values. All words \( W_i \) need \( \sum_{f=1}^{\tau} n_f \cdot f \cdot \log \tau = \log \tau \sum n_f \cdot f = O((n/\tau) \log \tau) \) because \( \sum n_f \cdot f = O(n/\tau) \). \( \square \)