Enabling Efficient Random Access to Hierarchically-Compressed Data

Feng Zhang*, Jidong Zhai†, Xipeng Shen‡, Onur Mutlu§, Xiaoyong Du*
*Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China
†Department of Computer Science and Technology, Tsinghua University, BNRist
‡Computer Science Department, North Carolina State University
§Department of Computer Science, ETH Zürich
fengzhang@ruc.edu.cn, zhaijidong@tsinghua.edu.cn, xshen5@ncsu.edu, onur.mutlu@inf.ethz.ch, duyong@ruc.edu.cn

***Abstract*—Recent studies have shown the promise of direct data processing on hierarchically-compressed text documents. By removing the need for decompressing data, the direct data processing technique brings large savings in both time and space. However, its benefits have been limited to data traversal operations; for random accesses, direct data processing is several times slower than the state-of-the-art baselines. This paper presents a set of techniques that successfully eliminate the limitation, and for the first time, establishes the feasibility of effectively handling both data traversal operations and random data accesses on hierarchically-compressed data. The work yields a new library, which achieves 3.1× speedup over the state-of-the-art on random data accesses to compressed data, while preserving the capability of supporting traversal operations efficiently and providing large (3.9×) space savings.**

I. Introduction

Text analytics is the process of analyzing text documents to discover useful information, draw conclusions, and assist decision-making. It is important in many domains, from web search engines to analytics in law, news, medical records, system logs, and so on.

Text analytics is fundamentally based on two types of operations on text, *traversal operations* and *random accesses*. *Traversal operations* traverse the entire text corpus. Examples include word count, inverted indexing, sequence count, and so on. Text clustering, for instance, often works on the result of inverted indexing; deep learning (e.g., LSTM [1] for translation) often works on the results of word embedding. Both inverted indexing and word embedding are traversal operations.

Random accesses, on the other hand, require visiting arbitrary locations in a text document. They are no less common than traversal operations. Examples include searching for a particular word, extracting a segment of content, and counting the frequency of a particular word or phrase.

Both kinds of operations face efficiency challenges for large datasets. One way to deal with large datasets is compression. Traditional compression methods, however, save storage space but increase data processing time, as the data must be decompressed before it can be processed.

Some recently-proposed compression techniques (e.g., Succinct [2]) try to avoid the need for data decompression before processing. However, these methods are designed for random accesses and do not work efficiently on traversal operations [2], [3].

A recent promising technique [3], [4] leverages hierarchical compression (i.e., the Sequitur algorithm [5]) to enable efficient traversal operations directly on compressed data without requiring decompression. The corresponding technique, TADOC (Text Analytics Directly on Compression), yields significant savings in both space (10×) and time (2×) for traversal operations [5].

Unfortunately, the benefits of TADOC disappear in the presence of random data accesses, due to the hierarchical compressed format of the data. A simple search for a particular word, for instance, becomes a time-consuming graph traversal: it takes seven seconds on a hierarchically-compressed two-gigabyte data using TADOC, five times longer than a simple sequential search on the original uncompressed data.

To avoid such slow handling of random accesses, TADOC needs to decompress the data first. Once data is decompressed, space saving benefits disappear. Recompression after random accesses is not a satisfying solution due to the long compression time. TADOC, for instance, takes over 20 hours to compress a 300GB dataset [3].

Moreover, TADOC does not support cases where new content is being continuously added to the dataset, while Succinct [2] supports only *append* operations to add new content to a dataset.

As such, two important open questions are 1) whether random accesses to hierarchically-compressed data can be made efficient and 2) whether the limitations on compressed dataset updates can be eliminated. Positive answers to the questions would eliminate the last major barriers for practical adoption of direct text analytics on compressed data.

This paper presents our solution, which consists of two major technical innovations. Our first innovation is a range of carefully designed indexing data structures. Our design enables reusability across analytics operations, and strikes a good balance between space cost and efficiency through these indexing data structures. Our second innovation is a set of algorithmic optimizations that enable random accesses to work efficiently on compressed data. These optimizations help maximize the performance of random data accesses by effectively leveraging the indexing data structures, incremental updates, recompression, and graph coarsening. We implement our techniques on TADOC, and show that they enable TADOC to achieve 3.1× speedup over the state-of-the-art (Succinct [2]) on random data accesses over compressed data, with or without continuous data growth. Our solution, at the same time, preserves 1) TADOC's unique capability of efficiently supporting traversal operations on compressed data and 2) most of TADOC's space reduction benefits, achieving 3.9× space savings compared to the original compressed datasets.

Overall, this work makes the following contributions:

- For the first time in literature, it provides a feasible and effective method for enabling efficient random access on hierarchically-compressed data.
- It delivers the first solution that can efficiently support direct text analytics on compressed data for both random accesses and traversal operations.
- It identifies five common types of random accesses in text analytics via analysis of a set of real-world text analytics workloads, and proposes a collection of techniques to efficiently support these operations on hierarchically-compressed data.
- It compares our techniques with the state-of-the-art, demonstrating its benefits in eliminating the last major barrier against the practical adoption of direct text analytics on compressed data.

II. Background

This section provides background on hierarchical compression and the previous technique, TADOC [3], [4], which leverages hierarchical compression for direct processing on compressed data.

TADOC uses a lossless hierarchical compression algorithm called Sequitur [5]. This recursive algorithm represents a sequence of discrete symbols with a hierarchical structure. It derives a context-free grammar (CFG) to describe each sequence of symbols: A repeated string is represented as a rule in the CFG. By recursively replacing the input strings with hierarchical rules, Sequitur produces a more compact output than the original dataset. For a set of text files, TADOC first adds some unique splitting symbols (called splitters) between files to mark their boundaries, and then applies Sequitur to build a CFG. The CFG is often several times smaller than the original data. It can also be represented as a directed acyclic graph (DAG).

Figure 1 provides an example. Figure 1 (a) shows the original input data: there are two files, `file0` and `file1`, separated by `SPT1`, and *wi* represents a word. Figure 1 (b) presents the output of TADOC in CFG form, which illustrates both the hierarchical structure and the repetition in the original input. It uses `R0` to represent the entire input, which consists of two files, `file0` and `file1`, represented by `R1` and `R2`. The two instances of `R2` in `R1` reflect the repetition of "`w1 w2`" in the substring of `R1`, while the two instances of `R1` in `R0` reflect the repetition of "`w1 w2 w3 w1 w2 w4`" in `file0`. The output of TADOC can be visualized with a DAG, as Figure 1 (c) shows, where edges indicate the hierarchical relations between rules. TADOC uses dictionary encoding to represent each word and rule with a unique non-negative integer, as shown in Figure 1 (d). It stores the mapping between integers and words in a dictionary. It assigns each rule a unique integer ID that is no smaller than `N` (`N` is the total number of unique words in the dataset; integers less than `N` are IDs of the words in the dictionary). Figure 1 (e) shows the CFG of Figure 1 (b) in numerical form.

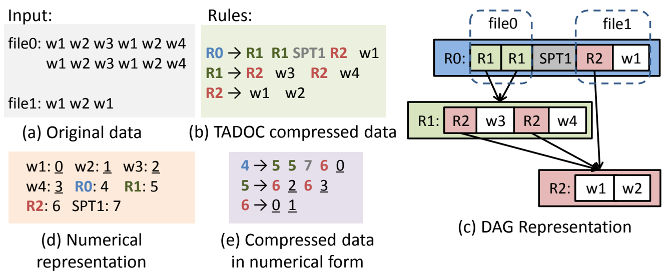

Fig. 1. A compression example with TADOC.

TADOC focuses on traversal operations in text analytics; it employs graph traversal on the DAG for those operations. We use `word count` as an example to illustrate how TADOC works. As Figure 2 shows, TADOC traverses the DAG in a bottom-up manner, counting the frequency of each word in each node it visits and the frequency of the words in the node's children. For example, when processing `R1` in Figure 2, TADOC counts `w3` and `w4` locally, and obtains the frequency of `w1` and `w2` by multiplying their frequencies in `R2` by the number of appearances of `R2` in `R1`. The traversal starts from leaf nodes and stops when it reaches `R0`. By leveraging the hierarchical structure of the compression format, TADOC avoids repeatedly counting text segments that appear many times in the input dataset, and hence can achieve significant time savings for traversal operations besides space savings for storing the dataset.

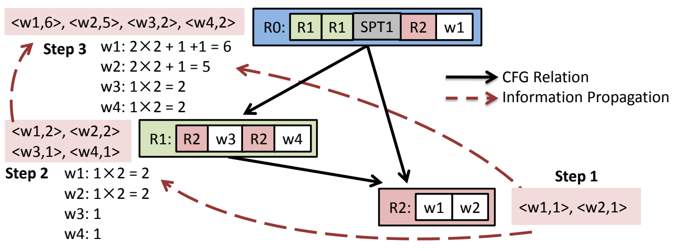

Fig. 2. An example of counting word frequencies with TADOC.

Although TADOC provides good performance, it is not efficiently applicable to all text analytics queries. Specifically, such compression-based analytics techniques do not support random accesses. First, hierarchical compressed data processing organizes data into a DAG, targeting only tasks that can be efficiently transformed into a DAG traversal problem. Second, as discussed in [3], TADOC is designed for datasets that are repeatedly used for many times without changes; when users want to perform insertion of new content, they would need to perform decompression first, and then recompress the data after the insertion of the new content. Compression with Sequitur takes a lot of time: 20 hours for compressing a 300GB dataset [3]. In this work, we aim to provide solutions to overcome these limitations.

III. Operations to Support and Challenges

This section first describes the types of important random accesses that we have identified in our workload analysis, and then discusses the challenges for supporting such random access types on hierarchically-compressed data.

A. Operations to Support

To identify the most important random access operations to support for text analytics, we have surveyed a set of domains where text analytics is essential, including news, law, webpages, logging, and healthcare. Our exploration leads to the following observations:

- Many uses of these text datasets involve several basic operations, `search`, `extract`, and `count`. For instance, in the news domain, data analysts locate relevant news events together to analyze their relationships by searching certain keywords [6]; in legal affairs, people may search and extract useful content from a large collection of law records [7], [8]; for webpages, searching or counting specific words, and extracting certain content are common operations [9], [10].
- In many domains, datasets are subject to the addition of new content. For instance, as news is continuously produced every day, the latest news could need to be appended to the existing news datasets. Rapid addition of new content is also important in IoT systems, which are organized in a decentralized structure and where many large logs are generated everyday [11]. Similarly, in healthcare, as more medical records are produced for a patient, they may need to be inserted into the existing collection of medical records [12], [13] (assuming records from many patients are stored as a whole).
- Deletion or replacement, on the other hand, is *not* common in the domains we examined. These datasets usually consist of data that has long-term value. Even though, due to space constraints, some old content may get moved to some other storage (e.g., tape), deletion or replacement are not common.

Based on these observations, we identify the following five types of random accesses as the essential ones to support for text analytics (in addition to the traversal operations prior work has already covered [3]). It is worth noting that Succinct [2], another efficient query processing engine designed for performing fast random access on compressed data, supports a similar set of operations, *except* insertion (Succinct can insert data only via *append*, which is limited).

- *extract(file,offset,length).* This operation returns a string of a given length of content at the offset in the file.
- *search(file,word).* This operation returns the offsets of all appearances of a specific word in a given file.
- *count(file,word).* This operation returns the number of appearances of a specific word in a given file.
- *insert(file,offset,string).* This operation inserts the input string at the offset of the file.
- *append(file,string).* This operation appends a string at the end of the file, which is much simpler than *insert*.

To support these five types of random accesses, we observe several principles:

- *Locality.* As these operations are random accesses to a specific word or text segment, the provided support should avoid the traversal of the DAG to find the place of interest. Such support should offer the capability to quickly locate the specific places in the dataset to operate on.
- *Compatibility.* The developed support should not only enable TADOC to perform these operations efficiently, but also preserve the capability of TADOC to support efficient traversal operations. This principle implies that the basic data structure of TADOC (i.e., the DAG from Sequitur) should stay as the main representation of the compressed dataset.
- *User Transparency.* To use these supported operations, users should not need to be concerned about how to implement them in their compressed datasets, but simply invoke some existing module's APIs. This principle is important for the practical usability and adoption of the developed support.

Achieving these goals on hierarchically-compressed datasets requires overcoming multiple challenges, as we discuss next.

B. Challenges

1) Hierarchical Structure of the DAG: The first is the hierarchical structure of the compressed data. In the DAG of Sequitur, one node (which corresponds to one rule in the CFG) could have multiple parents that belong to different files. An example is node `R2` in Figure 1 (c). The node has two incoming edges, respectively from `R1` and `R0`. The edge from `R1` to `R2` comes from some elements in `file0`, while the else from `R0` to `R2` comes from some elements in `file1`. Now consider the case where a user needs to quickly count how many times a word appears in a certain file in the compressed dataset. Starting from the root of the DAG and traversing the entire DAG to locate the word is apparently inefficient. A natural way to increase efficiency is to build up index ahead of time, recording the relations between each word and each rule. However, that index is not going to solve the problem: even if we find out that `w1` appears in both `R0` and `R2` in Figure 1 (c), using the index, we still cannot tell how many times the word appears in `file2` as rule `R2` belongs to multiple files.

2) Uni-Directionality: Second, currently, TADOC is unable to traverse the DAG from other nodes except the root node, since the DAG is a uni-directional data structure. Even if an indexing data structure allows us to immediately locate a node of interest, it is difficult to identify the node's sister nodes in the same file, as there are no edges going back to the parent of a node; even if there were, the node of interest may have multiple parents that belong to different files. For example, when we begin traversing from `R2` in Figure 1 (c), we do not know which node to visit next.

3) Special Complexities on Insert: Third, the hierarchical structure of the DAG imposes special complexities on the *insert* operation. As listed in Section III-A, *insert* places a new string at some offset in a file. The first step of *insert* is to efficiently locate the rule that contains the offset. The second step is to insert the string at that location. However, each rule represents a repeated string that appears more than once; if we directly insert the content into the rule that has one appearance corresponding to the desired offset, the same string would be, at the same time, inserted at other offsets where the rule appears. For instance, suppose that one wants to insert "`w7`" right after the third word in `file0` in Figure 1. If we insert it directly into rule `R1` after "`w3`", the consequence would be that `file1` now has "`w7`" inserted twice, at each of the places where `R1` appears (as shown in the content of `R0`).

4) Tradeoff between Space Savings and Time Cost: The fourth challenge is the tradeoff between space savings and time cost. One advantage of supporting analytics on compressed data is that we can enjoy space-saving benefits and also high performance at the same time, as previous work [3] has shown. However, to support efficient random access, some indexing data structures may have to be added. If the new data structures incur large space overhead, the advantage of the technique will reduce. Therefore, an important challenge is how to design the new data structures such that they can maximize the processing speed while minimizing any negative effects on space.

IV. Our Solution

A. Design Overview

To address the challenges against efficient random access on hierarchically-compressed data, we develop a series of novel techniques. Figure 3 illustrates the main challenges and our solution techniques. The first technique supports local graph walks (or partial traversals) starting from any place of interest in the DAG. This technique is essential for the *extract* operation (Section IV-B). The second technique builds efficient indexes between words and offsets in the DAG, which capture the complex relations among words, rules, and offsets. These indexes are especially useful for *search* and *count* operations (Section IV-C and IV-D). The third technique supports incremental dataset updates on the hierarchical compressed data. This technique makes efficient *insert* and *append* operations possible (Section IV-E and IV-F). We further consider graph *coarsening* as an optimization to save space cost (Section IV-G3).

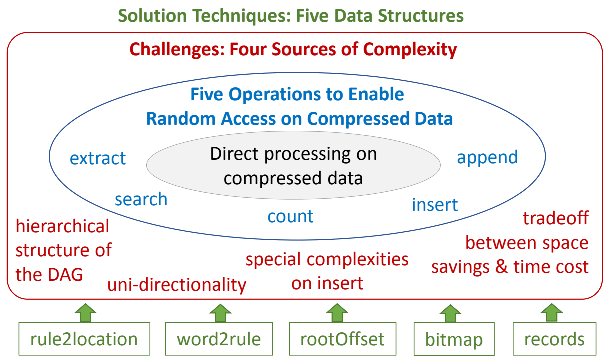

Fig. 3. Challenges against enabling random access to hierarchically-compressed data and our proposed solution techniques.

These techniques are not independent of each other; they work synergistically to address the various complexities in all types of random accesses. When designing the extra data structures required for each technique, we keep space overheads in mind and try to make a newly-introduced data structure useful for more than one type of operation. Specifically, we introduce five data structures, which we briefly explain below. We provide more detail on each data structure when we explain our techniques for each of the five random access types.

- *rule2location*. This data structure provides the mapping from each rule to the locations (the files and the offsets) where the string represented by the rule appear in the input data (Section IV-B).
- *word2rule*. This data structure provides the mapping from words to rules. For a given word, *word2rule* returns the set of rules the word appears in (Section IV-C).
- *rootOffset*. This data structure provides the offset of each element from the root rule (Section IV-E).
- *bitmap*. This data structure indicates whether or not an element in a rule has been changed (Section IV-E).
- *records*. This data structure stores the new content (Section IV-E).

These five data structures are designed to help address the challenges we described in Section III-B.

To address the first challenge of hierarchical structure of the DAG, *word2rule* and *rule2location* build the relation between words and offsets; given a word, we can quickly find its offsets in any document (we do not consider a potential *word2location* data structure due to its storage overheads).

To address the second challenge of uni-directionality, using the first three data structures, *rule2location*, *word2rule*, and *rootOffset*, we can perform local graph walks rather than traversing the graph from the beginning location for each random access.

To address the special challenge on *insert*, the bits in *bitmap* are used to indicate whether new content is added in each location, and the new content can be stored separately in *records*. These two data structures ease the handling of new content as a post-processing step.

Finally, to save space cost, these data structures are *selectively* stored. The largest data structure, *rule2location*, is not stored on disk but created on the fly when compressed data is loaded into memory.

We show an example of the relationships between these five data structures in Figure 4. *Rule2location* and *bitmap* are node-level data structures, which means that each node has its own instance of the two data structures. The other data structures are DAG-level data structures; i.e., there is only one instance of them for a given DAG. Data structure *rootOffset* is embedded in the root node. Among these data structures, only *rootOffset* is created on the fly while data is being loaded; the others are stored on disk. Section IV-F provides more details.

Next, we explain in detail how our proposed techniques support each of the random access types.

B. Extract

This operation extracts content directly from a compressed file. It is a basic operation required for reading data in

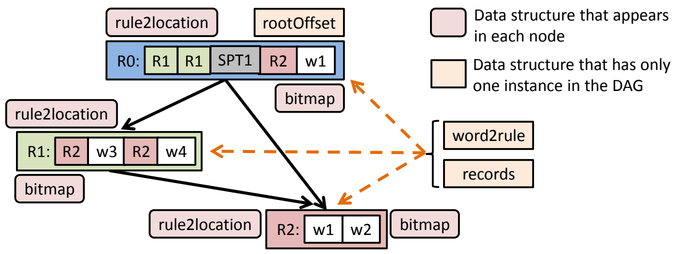

Fig. 4. Relationships between our new data structures.

compressed format for general types of analytics queries, since most queries first need to obtain the data.

Naive traversal-based approach. The most straightforward approach to designing the *extract* operation is to 1) traverse the DAG and record the length from the beginning, and 2) after reaching the starting location, extract the requested content. However, in this method, we need to search from the beginning of the root (`R0`) for each *extract* operation, which is prohibitively time-consuming. Therefore, we avoid such a design and instead develop two different approaches.

Our First Approach, a coarse-grained method. A more efficient method is to build indexes for the DAG. For each *extract* operation, we search the index of *rule2location* first, and then begin the traversal. However, a challenge blocks the partial traversal: the DAG does *not* provide pointers from children to parents (Section III-B2). To demonstrate this challenge, we use the example shown in Figure 5. Assume that we start the *extract* operation in `R4` of `file1`, we do not know which rule we should continue to traverse after we finish scanning `R4` due to *uni-directionality*. Note that only the root node does not have parents, and thus does not exhibit this challenge. Therefore, we first propose a coarse-grained method to keep the offset of each element at the root as our index. The core idea is to build a small number of indexes to save some of traversal operations in the DAG, and for an *extract* operation, we start traversal from an element in the root whose index is close to the required offset. For example, when the content in `R4` of `file1` is required, we can traverse directly from `R2` of `file1` in the root instead of the beginning of the root.

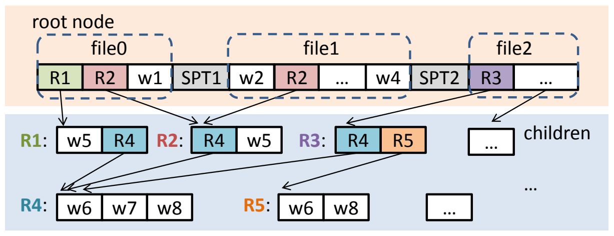

Fig. 5. An example of DAG representation for "`w5 w6 w7 w8 w6 w7` ...".

Such an indexing mechanism is an example of *range indexing*, which is a coarse-grained approach to *extract*. For a given *extract* operation, we can quickly locate the nearest starting position in the root, without traversing the DAG from the beginning. However, this approach has a drawback: unnecessary content from the index to the required offset still needs to be scanned, which can actually be avoided. Recall the example in Figure 5, and assume that we want to extract a string in `file1` and the starting position is in `R4`. Although the index in the first approach suggests us to traverse from `R2` of `file1` in the root instead of the beginning, we still need to scan the unrequested rule between the root and `R4`, which is `R2` in Figure 5. In real situations, there could be many such unnecessary rules between the root and the target rule, which causes significant time overhead.

Our Second Approach, a fine-grained method. To avoid the unnecessary time cost in our first approach, we need to build an index not only at the root, but also in subrules, so that we can start traversal in subrules; we call this fine-grained indexing. To tackle the challenge caused by the lack of pointers from children to parents, we build a data structure to indicate the relationships among rules.

Let us examine the challenge of how to maintain pointers from children to parents. As Figure 5 shows, a child such as `R4` can have multiple parents. The first challenge is how to record the right parent to visit after the child has been traversed. The parent may belong to different files (for example, in Figure 5, `R4`'s parent `R2` belongs to both `file0` and `file1`), which makes this more challenging. The second challenge is how to jump back to the right location in the parent. For example, in Figure 5, after `R2` has been processed in `file0`, we need to visit the third element (`w1`) in the root node, not the beginning of the root. The third challenge is where to add the index data structures, as different rules may have the same starting offset. For instance, in Figure 5, both `R2` and `R4` have the same offsets in `file1`; how to organize the index is a problem.

Detailed Design of Our Second Approach. Based on the above analysis, we develop a new data structure, called *rule sequence*, to provide the ability to index from children to parents. To enable this optimization, we extract the relationship among rules into a sequence, as shown in Figure 6. We use the DAG in Figure 1 (c) of Section II for illustration, and assume that the length of each word is two bytes. For each file, we store the starting offset, and *start* and *end* locations as a *unit* in each rule, into the $ruleSequence$ data structure. When rule shifting (i.e., traversing across different rules) happens, this rule sequence provides the necessary information to enable the ability to index from children to parents, which enables us to traverse forward and backward freely at any location of the DAG. We store this data structure in memory.

This design provides the pointers from children to parents, which can help us perform extraction directly from a subrule. To extract a piece of content, we can use binary search among offsets to quickly locate the starting unit; then, after we locate the starting unit, instead of DAG traversal, we go through the related rules with the help of the $ruleSequence$ data structure until we obtain the required content.

Algorithm 1 shows our second approach (Approach 2) for *extract*. We first use binary search to locate the starting unit (denoted as *startUnit*) in line 2. Then, we traverse the rule sequence from *startUnit* until we reach the unit that covers the requested content, as shown in lines 4 to 6. The *adjust()* function in line 7 adjusts the offset and the starting location, because the requested offset $start$ may not exactly match the offset of the located unit ($startUnit$). For the units within

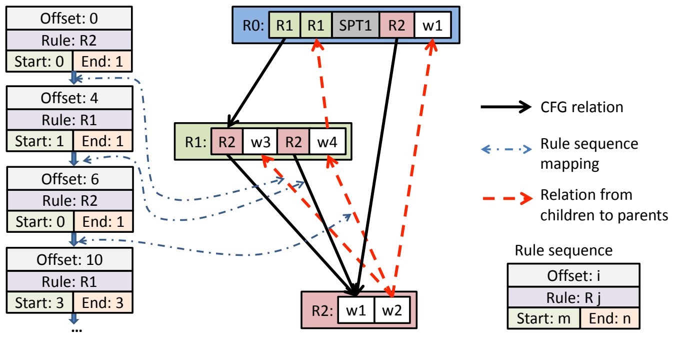

Fig. 6. Illustration of the *rule sequence* data structure for indexing. Assume the length of each word is two bytes.

the range from *startUnit* to *endUnit*, we sequentially add the elements from related parts of rules to the results, as shown in lines 8 to 12.

Algorithm 1 Extract len bytes from $start$ in file f (based on our second approach)

```
1:  function extract($f, start, len$)
2:      $startUnit = locate(ruleSequence[f], start)$
3:      $end = start + len$
4:      $endUnit = startUnit$
5:      while $ruleSequence[f][endUnit].end<end$ do
6:          $endUnit++$
7:      $adjust(start, end, startUnit, endUnit, ruleSequence[f])$
8:      for each unit $i$ in $ruleSequence[f]$ from $startUnit$ to $endUnit$ do
9:          $startElement = ruleSequence[f][i].start$
10:         $endElement = ruleSequence[f][i].end$
11:         for each element $j$ in $rule[i]$ from $startElement$ to $endElement$ do
12:             $output.push\_back(rule[i][j])$
13:     return $output$
```

C. Search

We provide an efficient design for the *search* operation, which returns the locations of occurrence of a given word. Different from the *extract* operation, the returned content of *search* may appear at any offset in a file. Therefore, it is necessary to create an efficient mapping from words to locations. A classic index for traditional document analytics is a mapping from words to the original documents, but such an index does *not* work in our hierarchical compressed format. The reason is that the document is represented by hierarchical rules in a DAG, and a rule can appear at multiple *different* locations in the original document. For example, in Figure 5, R4 appears at four locations in the original document, which indicates that the words in R4 have at least *four* related indexes. To build an efficient index in such a situation, we need to build the relations from words to rules, and then consider how to build the mapping from rules to locations, which is a complex *two-step* mapping instead of directly building the mapping from words to locations. On the other hand, the hierarchical representation also brings opportunities: a rule can be reused in many locations of the original document, so we can leverage such redundancy to build efficient indexes for the *search* operation.

Our Approach. Recall the data structures of *word2rule* and *rule2location* from Section IV-A. We can reuse these data structures to obtain the locations of a given word. First, we obtain the rules that contain the requested word via *word2rule*. Second, we use *rule2location* to calculate the exact offsets of the requested word in a file. Our detailed design follows.

Detailed Design. We show the pseudo-code of our *search* operation in Algorithm 2. *Search* provides the offsets of a given word in a given file. The data structure *word2rule* contains the mapping from words to rules, so *search* first checks the related rules for the word, which avoids unnecessary traversal. If the returned rule is the root, we need to traverse its elements via file splitters (lines 3 to 11), because only the elements in file f are necessary. During the traversal, we do not need to go into the subrules in root, but only add the length of these subrules to the offset (line 11). If the returned rule is not the root (lines 12 to 25), we need to scan the rule. Note that we need to verify whether or not the rule has been updated in the file before scanning. If so, we update the rule's location information (line 14). Each rule may have more than one location (location contains *file*, starting offset, and ending offset information). For example, in Figure 5, R4 has four locations: two in file0, one in file1, and one in file2. During rule scanning, we store the rule's starting offset in *offsetTmp* first, and then when we locate the word, we add the word's local offset in the rule to the locations of the elements in *offsetTmp* (lines 22 to 23). Finally, we examine the *records* data structure for further processing (line 26), since the new content added by *insert* or *append* may also contain the requested word (detailed in Section IV-E and IV-F).

Algorithm 2 Search $word$ in File f

```
1:  function search($f, word$)
2:      for each $i$ in $word2rule[word]$ do            ▷ $i$ is a rule
3:          if ($i == root$) then
4:              $rootStart = splitLocation[f]$
5:              $rootEnd = splitLocation[f+1]$
6:              $offset = 0$
7:              for each element $k$ from $rootStart$ to $rootEnd$ do
8:                  if ($k == word$) then
9:                      $output.push\_back(offset)$
10:                 else
11:                     $offset+ = length(k)$      ▷ $k$ can be a word or a rule
12:         else
13:             set $offsetTmp$
14:             $checkUpdateSearch(f, i)$
15:             for each element $j$ in $rule2location[i]$ do
16:                 if ($j.file == f$) then
17:                     $offsetTmp.push\_back(j.start)$
18:             $offset = 0$
19:             if ($offsetTmp.size$) then
20:                 for each element $m$ in $rule[i]$ do
21:                     if ($m == word$) then
22:                         for each element $loc$ in $offsetTmp$ do
23:                             $output.push\_back(loc + offset)$
24:                     else
25:                         $offset+ = length(m)$▷ $m$ can be a word or a rule
26:     $checkRecords4Search(records, f, word, output)$
27:     return $output$
```

Optimizations. We perform optimizations to make Algorithm 2 more efficient. We have mentioned two data structures in *search*, *word2rule* and *rule2location*, where *word2rule* is relatively simple. We describe the optimization of *rule2location*, which involves index mapping and storage format optimizations. The original index format for each rule is shown in Figure 7. The first element, *total*, stores the number of entries for a rule, and each entry contains three elements: *file_i*, *start_i*, and *end_i*, where *file_i* denotes the file ID the rule belongs to, and *start_i* and *end_i* denote the starting and ending positions of the rule in *file_i*. Because each rule may appear at different locations across different files, the number of entries can be large. To save space, we provide two optimizations. First, a rule may appear many times in one file, so we do not need to store *file_i* many times; instead, we store *file_i* once, and then follow the number of entries (*start_i* and *end_i*) in the file. Second, the length of a rule is fixed, so we do not need to store both *start* and *end* for all entries; instead, we store the length of the rule as *length*, and store only the starting location for each entry. Besides these optimizations, *coarsening*, an optimization technique that reduces the number of rules, also helps to make indexes more compact, as we discuss in Section IV-G3.

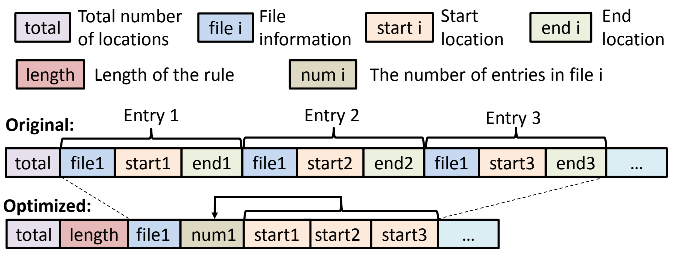
total Total number of locations
file i File information
start i Start location
end i End location
length Length of the rule
num i The number of entries in file i
Original:
Entry 1
Entry 2
Entry 3
total file1 start1 end1 file1 start2 end2 file1 start3 end3 ...
Optimized:
total length file1 num1 start1 start2 start3 ...

Fig. 7. Illustration of *rule2location* optimization.

D. Count

In this part, we discuss the design and insights of the *count* operation, which counts the occurrences of a certain word in a file. It differs from word-counting in prior work [3], [4] which, via traversal-type processing, counts the frequencies of *all* words. We develop two approaches to the implementation of *count*.

Our First Approach, the basic method. Given our design for the *search* operation in Section IV-C, we can easily develop *count* based on *search* with little change: *count* does not need to store the offsets for a word; instead, it only counts the occurrences of a given word, so we discard the offset information for the word. We regard this design as our basic method, and we predict that it has similar performance to *search*. This basic method uses the data structures in *search*, where offset information is unnecessary for the purposes of *count*, so we can further optimize the data structures for *count*.

Our Second Approach, the optimized method. To optimize the operation of *count*, we first review the data structures *word2rule* and *rule2location*. If we can obtain all the necessary information for *count* from these two data structures, we then circumvent the DAG traversal overhead. Recall that the goal of *word2rule* is to maintain a rule set for each word. We can also integrate the frequency of words to each rule in this set, so the new format of *word2rule* is: $< word_i, set < (rule_a, freq_a), (rule_b, freq_b), ... >>$, where $freq_j$ refers to the frequency of $word_i$ in $rule_j$. Next, with the help of *rule2location*, we can quickly obtain the rule frequency in each file (as "*num i*" in Figure 7 shows). In detail, to count a word in a given file, first, we obtain the word's local frequencies in the rules where it appears. Second, for the rules where the word appears, we obtain their rule frequencies in the given file, and multiply the rule frequencies with their associated local word frequencies. Third, the summation of the multiplication results is the required word count. For example, in Figure 5, we can directly obtain the word count for `w5` in `file0` by accumulating its word frequency in `R1` and `R2` using *word2rule* and *rule2location*, without requiring a DAG traversal.

Detailed Design of Our Optimized Method. Algorithm 3 shows our optimized algorithm for *count*. The data structure *ruleFreq* stores the rule frequency in each file, and its format is $< rule_i, set < (file_a, freq_a), (file_b, freq_b), ... >>$, where $freq_j$ refers to the frequency of $rule_i$ in $file_j$. Because the *root* rule contains the file splitters (as shown in Figure 5), we need to go through the root within the related file range to count the specific word, as lines 3 to 8 show. Finally, we examine the *records* data structure for further processing (line 13) so that we can consider the newly-added content (Section IV-E and IV-F). In Section VI-E, we compare our first approach, the basic method (based on *search*), and our second approach, the optimized method (Algorithm 3) in detail.

Algorithm 3 Count $word$ in File f

```
1:  function count($f, word$)
2:      for each $i$ in $word2rule[word]$ do                ▷ $i.rule$ is a rule
3:          if ($i.rule == root$) then
4:              $rootStart = splitLocation[f]$
5:              $rootEnd = splitLocation[f+1]$
6:              for each element $k$ from $rootStart$ to $rootEnd$ do
7:                  if ($k == word$) then
8:                      $output++$                          ▷ $output$ is the result
9:          else
10:             $localWordFreq = i.freq$
11:             $LocalRuleFreq = ruleFreq[i.rule][f]$
12:             $output += localWordFreq * LocalRuleFreq$
13:     $checkRecords4Count(records, f, word, output)$
14:     return $output$
```

E. Insert

The *insert* operation has the highest complexity among the five operations, because it changes data at an arbitrary location. We have considered a variety of design options, but each approach leads to several concerns.

The first option is to insert content directly into the DAG, which looks simple and straightforward. However, the first challenge is that, to be consistent with previous TADOC-based applications, we should not involve new types of data structures directly in the DAG; that is to say, we should still use the previous data structures in TADOC (*words*, *rules*, and *splitters*) to change the DAG. Unfortunately, because each rule can be reused more than once, i.e., a rule can appear at several offsets in the original file, we need to copy the rule that requires insertion to a new rule, and then insert content into the new rule. The second challenge is that, copying only one rule is not enough; the parent of the rule also needs duplication if it appears more than once. For example, in Figure 5, if we plan to insert a string in R4 from file1, we 1) need to duplicate R4 to a new rule where we insert the string; 2) next, need to duplicate its parent, R2, to a new rule, and 3) then change the new R2 to point to the duplicated R4. A similar process is repeated for all parents of the changed rules, until this recursive duplication process reaches a parent that appears only once. Hence, if the inserted rule has parents in multiple layers, this duplication process can incur large time and space overheads. Therefore, we abandon this design option.

The second design option is to perform decompression first, insert the content to the file in the decompressed format, and then perform compression. However, this method also has several drawbacks. First, the decompression and recompression processes have a large time cost when insertions are frequent. Second, the decompressed file size, which is the original file size, could be very large, and the machines that conduct this operation may not have enough space for such decompression. Therefore, we also abandon this option.

Our Approach. For the aforementioned reasons, our new design stores the newly-inserted content into a separate data structure (called *records*, which consists of *record* instances) instead of performing in-place insertion.

This design must address three complexities. The first complexity is how to indicate an insertion in the DAG. To solve this complexity, we introduce a *bitmap* data structure, where a bit corresponds to an element (an element could be a word or a rule) in the DAG, "1" indicating an insertion and "0" not. The second complexity is how to represent an insertion in a rule that appears at several locations. For example, in Figure 5, R4 has two locations in file0, one location in file1, and one in file2. We need additional information to indicate an insertion at a file offset in this case. To address this complexity, we store in the *record* data structure the starting offset of the rule along with the location in that rule where the insertion happens. This data structure provides the precise address of the inserted content in the DAG. The third complexity is how to handle *multiple* insertions at the same location in the DAG. To tackle this complexity, we add a pointer data structure "ptr" in *record*, which organizes all records inserted at the same location into a linked list. The structure of *record* is as follows.

The Record Data Structure

```
struct Record{
  int fileID; // file, such as file1
  int fileOffset; //file offset to insert, such as 100
  int ruleID; // the rule ID to insert, such as 0
  int ruleLocation; //the inserted location, such as 2
  int replaceWord; //the replaced word, such as w2
  string content; //content string
  int ptr; //the recordID inserted at the same place. Default is -1
  int ruleStartOffset; //the starting offset of the rule to insert, such as 0
};
```

With the data structure recording the necessary information, *insert* operates as follows. It first finds the offset in the first and second steps, which uses the same way as in *extract*. It then sets the corresponding *bitmap* and inserts the content into the records. Finally, it updates the *rootOffset* buffer as the newly-inserted content may change the starting offsets of some rules.

Insert Process

Let G be the graph representing compression results. Conduct insert(f,offset,string):

(1) Locate the element via "f" and "offset" in root. If it is a word, go to step (3).
(2) Traverse the rule to the location at "offset".
(3) Insert the "string" to "records", set the related bit to true, and add a pointer to the record in the DAG.
(4) Update "rootOffset".

F. Append

The *append* operation also changes data, but it is much simpler than *insert*, because the new content needs to be appended exactly at the *end* of a file. To help quickly find the end of a file for appending, in our design, when loading the compressed data, we record the last location of each file of the DAG in another buffer. For this purpose, we use the same data structure as in *insert* for the new content.

As our implementation makes no direct changes to the DAG, it ensures that other analytics, including the traversal operations [3], [4], can efficiently work on the DAG as usual. A post-processing step is needed to process the newly-inserted content. As the new content is stored in *records* without compression, the post-processing can be easily implemented by leveraging the *bitmap* and *records* data structures. Section VI-D evaluates the performance impact of the post-processing step on traversal operations.

G. Discussion

1) Recompression and Effect on Other Operations: For both *insert* and *append*, by default, the newly-added content is *not* compressed. When there is enough added content (the threshold is customizable by users), recompression can be invoked to compress all the old and new content together. Ideally, recompression should happen when the system is idle to avoid the performance impact of long recompression time while keeping the benefits of compression. The threshold to trigger recompression of data to incorporate the new data into the DAG (called *recompression frequency*) depends on the usage scenario and system settings. For instance, if new data arrives fast and the system has a lot of idle time and compute resources, recompression could happen more frequently; otherwise, it could happen less frequently. The use of parallel compression [14] can help reduce the compression time and find the best recompression frequency. In our experiments, for evaluation purposes, we use a simple policy as follows: Recompression happens when the size of the *records* data structure equals the size of the compressed data. How to determine the best recompression frequency for an arbitrary practical setting is a research topic that is worthy of future exploration.

2) Summary of Data Structures: Table I summarizes the data structures we use to support the five random access operations. The data structures in the third column are loaded into memory from disk, while the data structures in the last column are generated during data loading. *Extract* and *count*, as discussed in Section IV-B and IV-D, can be implemented using two different approaches, which use different data structures.

TABLE I
SUMMARY OF OUR DATA STRUCTURES.

Operation	Version	Data Structures	
		LoadedFromDisk	**GeneratedInMem**
extract	Approach1	DAG/dictionary/rule2location	
	Approach2	DAG/dictionary	ruleSequence
search		DAG/dictionary/rule2location/word2rule	
count	Approach1	DAG/dictionary/rule2location/word2rule	
	Approach2	dictionary/word2rule/ruleFreq	
insert		DAG/dictionary/bitmap/records	rootOffset
append		DAG/dictionary/bitmap/records	

3) Space Considerations: As stated in Section IV-A, we introduce five additional data structures to efficiently support random accesses. For *word2rule*, we already presented its optimized format in Section IV-D. For *rule2location*, we have shown its optimization in Figure 7 of Section IV-C. We have also illustrated the *records* data structure in Section IV-E. The other two, *rootOffset* and *bitmap*, are simple and straightforward. Among the five data structures, *rule2location* usually has the largest size. We found that rather than storing it on disk, it is better to build *rule2location* on the fly while loading the compressed data. The other data structures are stored on disk. To further save space for these five data structures, we employ an optimization called *coarsening* [3]. Coarsening merges some close-to-leaf subgraphs in the DAG to ensure that each leaf node contains at least a certain number of elements. It reduces the number of rules, and hence the size of our additional data structures. We analyze its effect on both space and performance in Section VI-E.

V. IMPLEMENTATION

We integrate our implementation of the support for the five random access types into the CompressDirect (CD) [3] library. In the new library, each operation is a separate module. *Search* module returns offsets of a certain word; *count* module counts the appearances of a given word; *extract* module extracts a piece of content; *insert* module performs insertions; *append* module appends data at the end of the dataset. For each of these modules, we implement sequential and distributed versions. The sequential version uses C++ and the distributed version uses C++ and Scala in the Spark environment [15]. In addition to these five modules, we also integrate a preprocessing stage to generate the necessary data structures, such as *word2rule*, *rule2location*, *rootOffset*, *bitmap*, and *records*.

VI. EVALUATION

Focusing on the five types of random access operations listed in Section IV, we evaluate the efficacy of the proposed support, in terms of both time and space savings. We report performance in both single-node and distributed environments.

A. Methodology

The baseline method we compare to is Succinct [2]. Succinct is the state-of-the-art method that supports random access on compressed data. It adapts compressed suffix arrays [16], [17] for data compression. As it is designed specifically for such operations, it achieves the highest speed on random accesses (but it is weak in efficiently supporting traversal processings). Our comparisons to Succinct examine whether our proposed support can make TADOC deliver comparable performance to Succinct on random access operations. If so, that would validate the promise of our techniques in making our expanded CD library the first library that efficiently supports both traversal and random access operations on hierarchically-compressed texts. Because Succinct does not have an *insert* operation, we use the *insert* function in the C++ string class as our baseline for *insert*.

Our method is denoted as “CD”. As "CD" is based on TADOC, we keep the inputs the same as those to TADOC [3], where text documents are first compressed with Sequitur and then compressed with Gzip [18]. During evaluation, our method first recovers the Sequitur-compressed result by undoing the Gzip compression, and then applies our direct processing mechanisms on Sequitur-compressed data. Our measured time includes both the time to recover Sequitur results and the processing time required for the random access operations. Note that even though preprocessing (such as data recovery) takes time, e.g., 41 seconds for dataset `A`, it is not a concern in practice, since its time cost is amortized over a large number of operations (*extract*, *search*, *count*, *insert*, and *append*) on the preprocessed data.

We automatically generate the inputs of the five types of random access operations. For *extract*, we pick random offsets in a file for extraction; the average length of extracted content is 64 bytes. For *search* and *count*, we randomly select a word from the vocabulary of a file. For *insert*, the offset is also random, and the string to insert is composed of randomly picked words from the dictionary; the average length of an inserted record is 64 bytes. Our settings for *append* are similar.

Datasets. Our evaluation uses five datasets that were used in previous studies [3], [4], shown in Table II. The first three datasets, `A`, `B`, and `C`, are large datasets from Wikipedia [19], which are used for evaluation on clusters. These datasets are Wikipedia webpages that are based on the same webpage template but that differ in content. Dataset `D` is NSF Research Award Abstracts (NSFRAA) from the UCI Machine Learning Repository [20], which is used for evaluating a large number of small files. Dataset `E` is from the Wikipedia database [19].

TABLE II
DATASETS ("SIZE" IS OF THE ORIGINAL DATASETS).

Dataset	Size	File #	Rule #	Vocabulary Size
A	50GB	109	57,394,616	99,239,057
B	150GB	309	160,891,324	102,552,660
C	300GB	618	321,935,239	102,552,660
D	580MB	134,631	2,771,880	1,864,902
E	2.1GB	4	2,095,573	6,370,437

Platforms. For the distributed system experiments, we use our `Spark Cluster`, a 10-node cluster on Amazon EC2 [21], and process datasets `A`, `B`, and `C`. Each node has two cores operating at a frequency of 2.3 GHz, is equipped with 8 GB memory, and its operating system is Ubuntu 16.04.5. The cluster is built on an HDFS storage system. Our Spark version is 2.1.0 while our Hadoop version is 2.7.0. Random access operations are written in C++, and we connect the operations to Spark via Spark pipe(). For Succinct, we use its C++ implementation with some minor changes; we also connect it to the Spark system.

For the sequential system experiments, we use our `Single Node` machine and process datasets `D` and `E`. This machine is equipped with an Intel i7-8700K CPU and 32 GB memory, and its operating system is Ubuntu 16.04.6. We compare our C++ implementation to Succinct's C++ version with default parameters.

B. Performance

1) Large Datasets: Figure 8 shows the throughput results (in terms of operations per second) for large datasets `A`, `B`, and `C` on the Spark cluster. In general, the five random access operations experience much higher throughput with our technique over Succinct. *Search* and *insert* have relatively low performance, *extract* and *count* have medium performance, while *append* has the highest performance. The reason is that *search* and *insert* involve many data accesses and operations on a large memory space, in both the compressed suffix array representation of Succinct and our hierarchical compressed representation. *Count* and *extract* do not have such overhead; *count* can obtain all the necessary information from *word2rule* and *rule2location*, and *extract* concentrates on only local areas in the dataset. For *append*, because our representation contains the locations of the end of files, new content can be appended directly without the need for accessing other parts of the DAG.

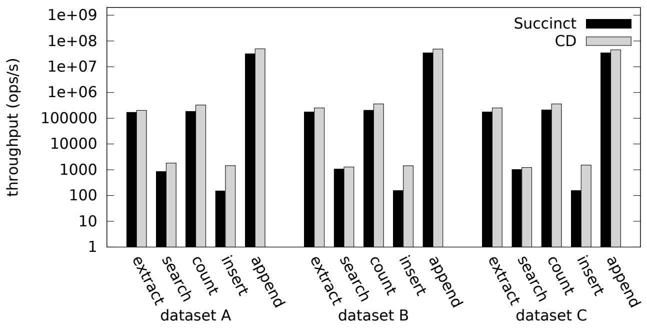

Fig. 8. Throughput for *extract*, *search*, *count*, *insert*, and *append* on different datasets on the Spark cluster.

Experiments show that our system consistently outperforms Succinct for the five operations on three datasets. For instance, CD achieves 234,764 *extract* operations per second on average, outperforming Succinct by 1.4×. CD achieves 1,443 *search* operations per second, outperforming Succinct by 1.5×. CD achieves 347,670 *count*, 1,462 *insert*, and 47,755,960 *append* operations per second, outperforming Succinct by 1.7×, 9.4×, and 1.4×, respectively. On average, the overall throughput of our proposed techniques is 3.1× of Succinct's throughput in a distributed environment.

Figure 9 shows the latency (in microseconds) of the five operations on large datasets on the Spark cluster. We define latency as the end-to-end time from when an operation starts until the time it finishes. The *append* operation has the lowest latency due to its simple algorithm; we store the appended content in a separate record and point to the end of a file, as described in Section IV-F. In contrast, the *search* and *insert* operations have relatively high latency, due to their complex interactions with the whole DAG. For the five operations, our system provides much lower latency than Succinct on most datasets: on average, CD reduces average operation latency by 17.5% over Succinct (*append* drags down CD's average performance).

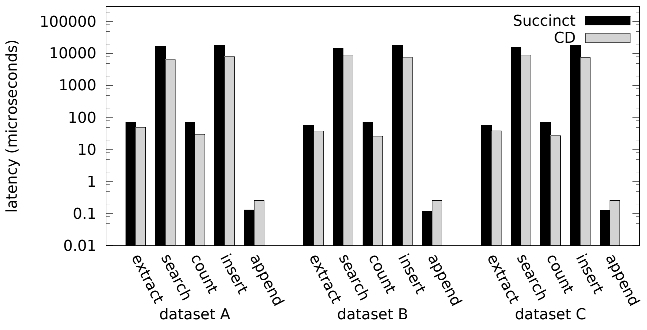

Fig. 9. Latency for *extract*, *search*, *count*, *insert*, and *append* on different datasets on the Spark cluster.

2) Small Datasets: Figure 10 depicts throughput results for small datasets on the `Single Node` machine. On average, our system provides 16× the throughput of Succinct. For *count* on dataset `D`, our system has lower throughput than Succinct. The reason is that dataset `D` contains a large number of files, which means that the data structure *ruleFreq* of dataset `D` is much larger than that of the other datasets. Obtaining the rule frequency for a given file with this data structure costs more time on dataset `D` than on the others, and our technique is less efficient than Succinct in this single case.

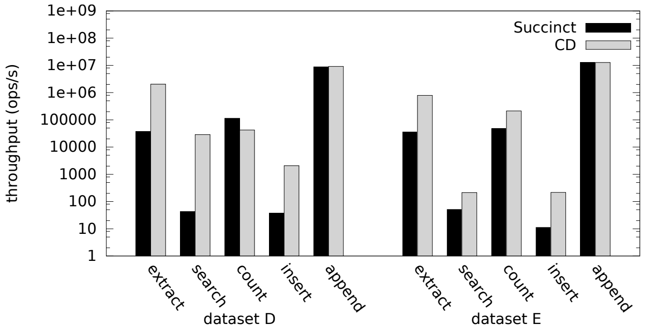

Fig. 10. Throughput for *extract*, *search*, *count*, *insert*, and *append* on different datasets on the Single Node machine.

C. Space Savings

We measure the space savings using the compression ratio metric, which is defined as *size(original)/size(compressed)*. The space-saving results are shown in Table III. CD improves on TADOC [3] via the new data structures, as mentioned in Section IV-A, to support random accesses. The compression ratio of the original TADOC is 6.5–14.1. The newly added data structures inflate the space, decreasing the compression ratio to 2.6–5.0. The average compression ratio we observe is 3.9, which is still much more compact than the 1.8 compression ratio of Succinct.

Among the data structures used in our evaluation, two data structures, *ruleSequence* and *rootOffset*, are created on the fly in memory when data is being loaded. Two data structures, *bitmap* and *records*, do not need to be stored on

TABLE III
COMPRESSION RATIOS.

Version	Dataset A	B	C	D	E	AVG
Uncompressed	1.0	1.0	1.0	1.0	1.0	1.0
Succinct [2]	2.2	1.7	1.6	2.9	2.2	1.8
Original TADOC [3]	14.1	13.3	13.1	6.5	11.9	11.8
CD	4.9	3.5	3.5	2.6	5.0	3.9

disk because initially no insertion happens. Therefore, the only data structures that incur disk storage cost are *rule2location*, *word2rule*, and *ruleFreq*, and their space breakdown in storage is shown in Table IV; *rule2location* occupies most of the space.

TABLE IV
SPACE BREAKDOWN FOR DIFFERENT DATA STRUCTURES (MB).

Data Structure	A	B	C	D	E
rule2location	4707	13427	26815	122	192
word2rule	433	1218	2442	15	14
ruleFreq	27	76	151	65	2.1

D. Traversal Operations on Added Content

Adding support for *insert* and *append* to direct processing on compressed data is an important contribution of this work. Using our support, previous traversal operations [3], [4] can still work on the updated dataset. Only a post-processing step is needed to process the newly added content. We implement a post-processing step for each of the traversal operations proposed in earlier work [3]. This section reports the measured time of such traversal operations in our system.

The total execution time consists of two parts: 1) DAG processing, which is the same as in previous work [3], and 2) post-processing, which processes the new content in *records*. Considering the size of datasets, in this experiment, we randomly insert 10,000,000 records into dataset A, 30,000,000 records into dataset B, 60,000,000 records into dataset C, and 400,000 records into both datasets D and E. Table V reports the fraction of time spent on post-processing in each of the six traversal data analytics workloads. The ratio ranges from 3.7% to 30.4%, confirming that with our method, TADOC can now effectively handle data insertion and *append* operations.

TABLE V
FRACTION OF TIME SPENT ON POST-PROCESSING.

	Fraction of time on each dataset (%)					
Application	A	B	C	D	E	AVG
Word Count	23.0	10.3	10.5	13.4	5.5	12.5
Sort	7.4	6.5	8.1	12.8	3.7	7.7
Inverted Index	21.1	13.8	17.1	10.8	4.8	13.5
Term Vector	20.6	9.8	9.1	8.7	4.8	10.6
Sequence Count	17.2	11.0	20.7	29.8	30.4	21.8
Ranked Inverted Index	13.4	7.3	13.4	22.3	29.7	17.2

E. Tradeoff between Performance and Space

1) Different data structures: The tradeoff between time and space is affected by the choices of data structures. In Section IV, we discuss two versions of *count* and *extract*. Table VI provides a detailed analysis of time and memory consumption of each version. In our evaluation, our Approach 2 to *extract* (Algorithm 1 in Section IV-B) achieves an average of 9,756× throughput improvement over that of Approach 1 in Section IV-B). Our Approach 2 to *count* (Algorithm 3 in Section IV-D) achieves an average of 70× throughput improvement over Approach 1 (the basic *count* version in Section IV-D) based on Algorithm 2. However, for *extract*, Approach 1 has smaller memory consumption; the reason is that Approach 2 generates *ruleSequence* during runtime, which consumes large memory space.

TABLE VI
THROUGHPUT AND MEMORY CONSUMPTION BREAKDOWN OF DIFFERENT IMPLEMENTATIONS OF *count* AND *extract*.

		Throughput (ops/second)		Memory (MB)	
Operation	Dataset	Approach 1	Approach 2	Approach 1	Approach 2
extract	A	75.7	201851.9	19942	43345
	B	78.0	251219.5	45324	111700
	C	85.8	251219.5	84176	216706
	D	51321.8	2040440.0	493	1469
	E	19.9	793624.0	1030	1937
count	A	3244.0	324404.8	22725	9190
	B	3029.4	359302.3	53170	14762
	C	3121.2	359302.3	99874	23056
	D	28476.6	42318.7	550	303
	E	13318.5	212723.0	1135	467

2) Coarsening: The tradeoff between time and space is also affected by coarsening. Table VII reports the effects of coarsening in more depth. Coarsening greatly reduces the storage size, especially for the data structures related to rules. Note that coarsening does not decrease the size of the DAG (the rule with a small size needs to be merged to all its parents, thereby causing redundancy), but it greatly reduces the size of the data structures related to rules, thereby reducing the overall storage size. As the second column in Table VII shows, the space savings from coarsening is over 62% for all datasets. An expected effect of coarsening is that as each leaf node becomes larger, some more time may be needed for locating a word or offset in the leaf nodes. The other columns in Table VII report the potential additional speedup our method could achieve if it does not use coarsening; we find that coarsening decreases performance, because more redundant content needs to be scanned after coarsening. Our implementation chooses to employ coarsening as coarsening provides a more desirable trade-off between space savings and speedup.

TABLE VII
STORAGE SAVINGS WITH COARSENING AND THE POTENTIAL SPEEDUP WHEN COARSENING IS NOT USED.

	Space	Potential Speedup without Coarsening (×)				
dataset	Savings	search	count	extract	insert	append
A	64.0%	3.2	1.3	1.4	2.4	1.1
B	62.5%	4.5	1.0	1.2	2.7	1.2
C	63.1%	4.9	1.0	1.1	2.6	1.3
D	65.1%	1.9	4.4	0.6	0.2	1.1
E	64.0%	3.8	8.5	0.8	2.4	1.1
AVG	63.7%	3.7	3.3	1.3	2.1	1.1

VII. RELATED WORK

To our knowledge, this work is the first to enable efficient support for both random access operations and traversal operations on hierarchically-compressed data. We overcome the limitations of TADOC [3], [4] in efficiently supporting random accesses. We do so by introducing a novel set of carefully-designed data structures and optimizations to support random access operations on hierarchically-compressed data.

We further add support for efficiently incorporating new data into a hierarchically-compressed dataset.

Sequitur is a well-known grammar-based compression algorithm [5], [22], [23]. It is first used for direct processing on compressed data by TADOC [3], [4]. Besides text analytics, Sequitur is used for various other purposes, such as improving data reference locality [24], dynamic hot data stream prefetching [25], analyzing whole program paths [26], [27], finding loop patterns in program analysis [28], XML query processing [29], and comprehension of program traces [30].

Succinct [2] is a high-performance query engine on compressed data that is designed for databases. Our work is orthogonal to Succinct in both implementation and applications. In terms of implementation, Succinct extends indexes and suffix arrays [31] as basic compression structures, while our work extends a hierarchical compression method, Sequitur [5]. In terms of applications, Succinct is designed for database queries while our work is designed for general text analytics. Importantly, Succinct [2] provides no mechanism to efficiently incorporate new data into a compressed dataset; our work provides a new design for efficiently doing so. The results in Section VI show that our method achieves much higher performance than Succinct on random access operations, while keeping TADOC's distinctive strength in supporting traversal operations. In contrast, Succinct supports arbitrary substring and regular expression searches, and broader data types; we leave such support as future work for our methods. The compression method used in Succinct is also employed in other studies [32], [33].

VIII. Conclusion

This paper presents a set of new techniques that enable efficient random access operations on hierarchically-compressed data, significantly expanding the capability of prior works on text analytics on compressed data. Altogether, our proposed techniques provide the first library that efficiently supports both traversal and random access operations directly on compressed text files, and by doing so, remove a major barrier against practical adoption of direct text analytics on compressed data.

IX. Acknowledgments

This work is partially supported by the National Key R&D Program of China (Grant No. 2017YFB1003103), National Natural Science Foundation of China (Grant No. 61802412, 61732014, 61722208), and Tsinghua University-Peking Union Medical College Hospital Initiative Scientific Research Program. This material is also based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-1717425 and CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Onur Mutlu is supported by ETH Zürich, SRC, and various industrial partners of the SAFARI Research Group, including Alibaba, Huawei, Intel, Microsoft, and VMware. Jidong Zhai, Xipeng Shen, and Xiaoyong Du are the corresponding authors of this paper.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.
[2] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *NSDI*, 2015.
[3] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *PVLDB*, 2018.
[4] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data," in *ICS*, 2018.
[5] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, 1997.
[6] B. Zhao and S. Vogel, "Adaptive parallel sentences mining from web bilingual news collection," in *ICDM*, 2002.
[7] A. B. Bepko, "Public availability or practical obscurity: the debate over public access to court records on the internet," *NYL Sch. L. Rev.*, 2004.
[8] P. A. Winn, "Online court records: Balancing judicial accountability and privacy in an age of electronic information," *Wash. L. Rev.*, 2004.
[9] S. Bao, J. Chen, L. C. En, R. Ma, and Z. Su, "Method and apparatus for enhancing webpage browsing," 2013.
[10] S. Lawrence and C. L. Giles, "Context and page analysis for improved Web search," *IEEE Internet Computing*, 1998.
[11] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. A. Lee, and J. Kubiatowicz, "The Cloud is Not Enough: Saving IoT from the Cloud," in *HotStorage*, 2015.
[12] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," *Health information science and systems*, 2014.
[13] R. H. Miller and I. Sim, "Physicians' use of electronic medical records: barriers and solutions," *Health affairs*, 2004.
[14] P. Jalan, A. K. Jain, and S. Roy, "Identifying Hierarchical Structures in Sequences on GPU," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, 2015.
[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, 2010.
[16] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.
[17] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, 2005.
[18] "Gzip," https://www.gzip.org/, 2019.
[19] "Wikipedia HTML data dumps," https://dumps.wikimedia.org/enwiki/, 2017.
[20] M. Lichman, "UCI machine learning repository," http://archive.ics.uci.edu/ml, 2013.
[21] "Amazon EC2," https://aws.amazon.com/ec2/, 2019.
[22] C. G. Nevill-Manning, "Inferring sequential structure," Ph.D. dissertation, University of Waikato, 1996.
[23] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *DCC*, 1997.
[24] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *PLDI*, 2001.
[25] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *PLDI*, 2002.
[26] J. R. Larus, "Whole program paths," in *PLDI*, 1999.
[27] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE*, 2003.
[28] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *ISPASS*, 2005.
[29] Y. Lin, Y. Zhang, Q. Li, and J. Yang, "Supporting efficient query processing on compressed XML files," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.
[30] N. Walkinshaw, S. Afshan, and P. McMinn, "Using compression algorithms to support the comprehension of program traces," in *Proceedings of the Eighth International Workshop on Dynamic Analysis*, 2010.
[31] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
[32] A. Khandelwal, R. Agarwal, and I. Stoica, "BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores," in *NSDI*, 2016.
[33] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica, "ZipG: A Memory-efficient Graph Store for Interactive Queries," in *SIGMOD*, 2017.