POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression

Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du

Abstract—Parallel technology boosts data processing in recent years, and parallel direct data processing on hierarchically compressed documents exhibits great promise. The high-performance direct data processing technique brings large savings in both time and space by removing the need for decompressing data. 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 article proposes a novel concept, orthogonal processing on compression (orthogonal POC), which means that text analytics can be efficiently supported directly on compressed data, regardless of the type of the data processing – that is, the type of data processing is orthogonal to its capability of conducting POC. Previous proposals, such as TADOC, are not orthogonal POC. This article presents a set of techniques that successfully eliminate the limitation, and for the first time, establishes the near orthogonal POC feasibility of effectively handling both data traversal operations and random data accesses on hierarchically-compressed data. The work focuses on text data and yields a unified high-performance library, called POCLib. In a ten-node distributed Spark cluster on Amazon EC2, POCLib 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.

Index Terms—Near orthogonal processing on compression, direct processing on compressed data, TADOC, orthogonal POC

1 Introduction

Text analytics refers to 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.

To deal with the worsening time and space pressure imposed by the rapid data growth, researchers have recently developed a new technique, called TADOC, to support text analytics directly on compressed data, without data decompression [1], [2], [3]. TADOC employs a hierarchical compression method, Sequitur [4], to compress text into a directed acyclic graph (DAG) (or, equivalently, a context-free grammar, CFG), and then conducts text analytics using graph traversal operations. Compared to the processing directly on the original dataset, TADOC saves over ten times storage and memory space while shortening the processing time by half.

Despite the promising results, TADOC suffers an important drawback, the lack of orthogonality—which relates with the goal of this current work, **orthogonal processing on compression**, or **orthogonal POC** in short. Orthogonal POC is a new concept introduced in this work. A compression-based analytic technique is an *orthogonal POC* technique if it supports efficient[1] data analysis directly on the compressed data, regardless of the type of the data processing—that is, the type of data processing is orthogonal to its capability of conducting POC.[2]

The previous proposal, TADOC, is not an orthogonal POC. It supports traversal kind of text analytics (e.g., word count, inverted indexing, sequence count) that traverse the entire text corpus, but cannot support efficient random data accesses—such as searching for a particular word, extracting a segment of content, and counting the frequency of a particular word or phrase; it cannot work in scenarios where new content is continuously inserting or appended to the dataset. On the other hand, there are some other techniques (e.g., Succinct [5]) that support random accesses to compressed data, but do not support the complex traversal processings that TADOC supports.

Being orthogonal is important for a POC, as it avoids the need for decompressing data in all circumstances. It is worth noting that once data are decompressed due to a task

• Feng Zhang and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China. E-mail: {fengzhang, duyong}@ruc.edu.cn.
• Jidong Zhai is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhaijidong@tsinghua.edu.cn.
• Xipeng Shen is with the Computer Science, North Carolina State University, Raleigh, NC 27695 USA. E-mail: xshen5@ncsu.edu.
• Onur Mutlu is with the Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. E-mail: omutlu@ethz.ch.

Manuscript received 29 May 2020; revised 26 May 2021; accepted 15 June 2021. Date of publication 29 June 2021; date of current version 23 July 2021.
(Corresponding authors: Jidong Zhai and Xiaoyong Du.)
Recommended for acceptance by J. Zhan.
Digital Object Identifier no. 10.1109/TPDS.2021.3093234

1. Here, being efficient means that the POC takes less time than the default processing on the original uncompressed data does.
2. In this definition, "orthogonal" means one property is perpendicular to—that is, independent of—another property.

1045-9219 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

that a POC does not support, most benefits of the POC disappear: no space savings anymore, and if the task involves data changes, for the data would need to be recompressed for POC to keep supporting other operations. Recompression takes time. In TADOC, for instance, the time to compress a 300GB dataset is over 20 hours [2].

This paper presents our work towards creating the first near orthogonal POC technique. A *near orthogonal POC* is a POC that works efficiently for most common operations. Ensuring a POC work for all operations is difficult as the kinds of operations are hard to enumerate. A *near orthogonal POC* can meet most practical needs. Even for specific operations that do not support currently, near orthogonal POC means that we can solve such issues by adding extra basic operations (detailed in Section 3.2).

We choose TADOC as the base for the development, for it is the only existing technique that supports complex traversal processings on compressed data. With TADOC, our objective reduces to expanding TADOC to support random data accesses without compromising its support of traversal processings.

The key challenges come from the hierarchical data representation in TADOC. Such a representation is a double-bladed sword. It makes traversal processings easy to support as graph traversals, but, at the same time, makes it hard to efficiently locate a particular place in the original data on the compressed format (detailed in Section 2). A simple search for a particular word, for instance, becomes clunky: It takes seven seconds on a compressed two gigabyte data on TADOC, 5 times longer than a simple sequential search on the original uncompressed data. The disparity quickly worsens as new texts get inserted into the original data.

To solve these challenges, we present **POCLib**, a unified near orthogonal **P**rocessing **On** **C**ompression **Lib**rary for text analytics. POCLib 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 develop POCLib based on TADOC, elevating TADOC into the first near orthogonal POC technique. Experiments show that POCLib enables TADOC to achieve 3.1× speedup over the state-of-the-art (Succinct [5]) on random data accesses over compressed data, with or without continuous data growth. POCLib, 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 introduces the concept of *orthogonal POC*, and delivers the first near orthogonal POC 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 together with TADOC, proposes POCLib to efficiently support these operations on hierarchically-compressed data.
- It compares POCLib with the state-of-the-art, demonstrating its benefits in creating the first near orthogonal POC solution for text analytics.

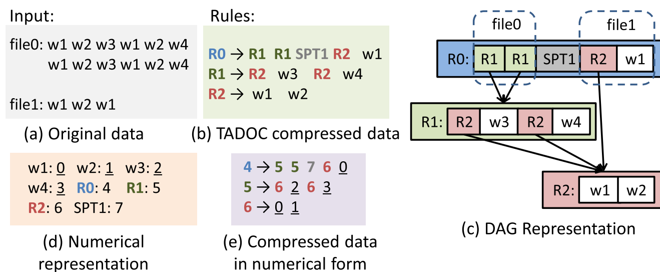


Fig. 1. A compression example with TADOC.

2 Background

This section provides background on hierarchical compression and the previous technique, TADOC [1], [2], [3], which leverages hierarchical compression for direct processing on compressed data.

TADOC uses a lossless hierarchical compression algorithm called Sequitur [4]. 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).

Fig. 1 provides an example. Fig. 1a shows the original input data: there are two files, `file0` and `file1`, separated by `SPT1`, and *wi* represents a word. Fig. 1b 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 Fig. 1c 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 Fig. 1d. 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). Fig. 1e shows the CFG of Fig. 1b in numerical form.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply.

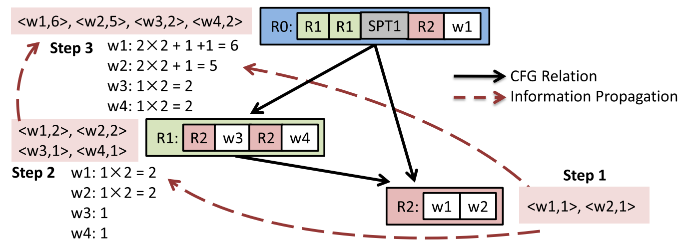


Fig. 2. An example of counting word frequencies with TADOC.

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 Fig. 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 Fig. 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.

TADOC Limitation. TADOC is not orthogonal POC. 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 [2], 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 [2]. In this work, we aim to provide solutions to overcome these limitations.

3 Orthogonal POC

Orthogonal POC supports efficient data analytics on compressed data for all types of data processing, with minimal impact on its performance. We describe the specification and rationale of orthogonal POC, informed by the characteristics of common data analytics. Next, we show our near orthogonal POC properties, since complete orthogonal POC is hard to realize. Finally, we show the benefits of our near orthogonal POC. Note that data analytics broadly includes multiple types of data, such as text, pictures, and voices. This paper mainly uses the text as an example for illustration.

3.1 Specification and Rationale of Orthogonal POC

This work introduces orthogonal POC for text analytics: *A compression-based analytic technique is orthogonal POC if it supports efficient direct processing of the compressed data, regardless of the type of the processing*. Orthogonal is defined as the persistence for various types of data analytics, and POC is defined as the efficiency of directly processing on compressed data. With an orthogonal POC, once data have been compressed, they can be used efficiently for both traversal and random accesses. Orthogonal POC aims to provide a uniform model for all aspects of data analytics, which covers the data structures and application behaviors. Orthogonal POC is characterized by the following three principles:

- *Analytics Orthogonality*. All types of data analytics, including future-proofed analytics, are supported. The universality, or orthogonality, means that data characteristics can be preserved across the compressed data representation and special mechanisms are defined to handle all analytics tasks users provide.
- *POC Efficiency*. A compression-based data analytics technology is POC efficient if and only if the POC takes less time and space than the default processing on the original uncompressed data. By efficient data reorganization and reuse, the POC achieves better locality and avoids unnecessary computation, thus achieving high efficiency.
- *Persistence Independence*. All analytics tasks must accept common analytics representation standards. That is, the programming interfaces provided to various data analytics are identical whether the data analytics are based on traversal operations or random accesses. These standards derive from common data analytics and are future proof.

3.2 Near Orthogonal POC

Developing a completely orthogonal POC is difficult as it is hard to enumerate all possible operations on texts. A more practical goal is to develop a *near orthogonal POC*, which can support most common operations in text analytics, both traversal operations and random accesses. Such a technique can already meet most practical needs.

To further develop a near orthogonal POC framework, we claim that the techniques should follow **LACT** (**l**ocality, **a**daptability, **c**ompatibility, **t**ransparency), which is a set of properties for near orthogonal POC shown as follows.

- *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.
- *Adaptability*. The POC system or the defined standards should derive the ability to adapt efficiently to various data analytics. The orthogonal POC could be open when unable to adapt to certain analytics tasks, and can solve such issues by adding extra basic operations reaching a wide range of applications.
- *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.,

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

the DAG from Sequitur) should stay as the main representation of the compressed dataset.

- *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.

How close is the near orthogonal POC proposed in the paper to a complete orthogonal POC? The gap between the near orthogonal POC and a complete orthogonal POC can be analyzed from the three principles of orthogonal POC mentioned in Section 3.1. First, instead of claiming *analytics orthogonality*, near orthogonal POC advocates *adaptability*, which means that the system should be open. Even certain analytics tasks are found unable to support currently, we can still solve such issues by adding additional operations. Second, instead of claiming *POC efficiency* from both space and time perspectives, near orthogonal POC emphasizes the *locality* during processing compressed data, in case of specific analytics tasks where the benefits of both time and space dimensions may not be fully guaranteed simultaneously. Third, instead of claiming persistence independence, near orthogonal POC emphasizes *compatibility* and *transparency*, which are more practical to current data analytics on compressed data.

3.3 System Benefits

Our strategy for creating a near orthogonal POC framework is to take TADOC as the base and develop new support for it to efficiently support the common random accesses in text analytics. The benefits come from three aspects. First, TADOC is successful at providing text analytics mechanism directly on compression; the applications already supported do not need to be changed. Second, TADOC already provides basic grammar-based data structures for compression, which can be reused. Third, a broad range of data analytics could be supported. The next section describes the list of important random accesses that we have identified in our investigation, and then discusses the challenges for supporting them on the hierarchically compressed data.

3.4 System Challenges

Developing a near orthogonal POC framework based on TADOC requires to handle three challenges, as listed below.

(1) *Efficiency*. As discussed in Section 3.1, orthogonal POC requires that the compressed data are processed in an efficient manner. First, TADOC is efficient to support traversal operations [1], [2], [3], but is hard to efficiently locate a particular place for random accesses on compressed format. For example, random accesses requires traversing the DAG from any nodes, but current TADOC design is uni-directionality, which provides pointers only from parents to children. Second, the efficiency lies in both time and space perspectives. However, to support random accesses, we need to add index data structures, which relates to the tradeoff between time and space. Third, data structures in TADOC, such as the rule formats, dictionary, and DAG organizations, should be reused for efficiency.

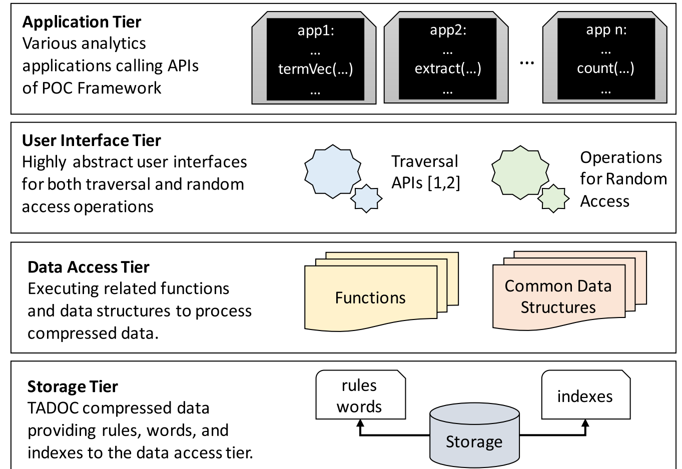


Fig. 3. Tier architecture of POCLib.

(2) *Integration*. As discussed in Section 3.3, we should integrate new functionalities to TADOC [1], [2], [3]. First, previous TADOC functions and data structures should not be changed since TADOC has already been applied to several text analytics tasks [2]. Second, new functions and data structures should be regarded as an extension to TADOC and be compatible with it. Third, previous TADOC techniques do not support data insertion, which is a common operation and should be involved in near orthogonal POC. All these limitations increase the difficulties of integrating random accesses to TADOC traversal operations.

(3) *API Design*. The first challenge lies in the API design for usability. First, the interfaces should be highly simplified, which can reduce user learning curve. Second, the number of APIs should be minimized, which means that the newly extended operations cannot be substituted efficiently for each other. Third, the APIs should be compatible with popular programming environments, because current data analytics applications could be implemented in various languages, such as JAVA, C/C++, and Python.

4 POCLib Framework

In this section, we first describe our general design of POCLib; we adopt a four-tier design and integrate random access techniques to TADOC. Next, we analyze the operations we should support for near orthogonal POC. Then, we show the system workflow of POCLib.

4.1 System Design

The architecture of POCLib is shown in Fig. 3, which includes four tiers. The first tier is the application tier, which calls the provided APIs to interact with POCLib. The second tier provides user interfaces. Our system provides two kinds of APIs: the APIs for traversal-based text analytics, which are borrowed from [1], [2], [3], and the APIs for random access operations, which are the target of this paper. The third tier is the data access tier. It includes the internal functions and common data structures, which come from two parts: traversal operations from [1], [2], [3] and random access that shall be discussed in Section 4.2. The fourth tier is the storage tier, which stores the TADOC-compressed data and can provide rules, words, and indexes to support

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

the data access tier. Since we integrate the graph operations from our previous work, TADOC [1], [2], [3], we need to measure the performance of these operations after integration into POCLib, which is detailed in Section 7.4.

Solutions to Challenges. We solve the challenges described in Section 3.4 during designing POCLib. For the first challenge in efficiency, we reuse the TADOC data structures and DAG representation in POCLib, and add loosely coupled index data structures to achieve the purpose of reasonably maintaining the space benefits of TADOC. For the second challenge of integration, we add extra data structures and additional functions to POCLib while keep the previous data structures and functions in TADOC unchanged. For the third challenge of POCLib API design, we analyze common text analytics and abstract five common operations to support (detailed in Section 4.2).

4.2 Operations to Support

To identify the most important random access operations to support for text analytics in POCLib, 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 is 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 POCLib (in addition to the traversal operations prior work has already covered [2]). It is worth noting that Succinct [5], 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).

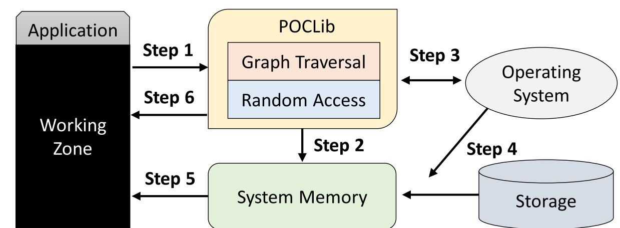


Fig. 4. System workflow.

- *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*.

4.3 System Workflow

We present the system workflow of POCLib in Fig. 4. In Step 1, users issue requests of traversal operations and random accesses to POCLib framework in their working zone of the application. In Step 2, POCLib looks for data in accordance with the user-issued commands in the system memory. If the related content can be found in the system memory, the system executes the user-provided requests (Step 5) and returns the successful status (Step 6); otherwise, it goes to Step 3. In Step 3, the POCLib framework operates with the operating system to locate the related compressed data. In Step 4, the compressed data are loaded into memory with proper data structures. In Step 5, POCLib executes the user commands and provides necessary content back to the working zone of the user application. In Step 6, POCLib returns the execution status back to the application.

In the rest of this paper, we focus on the detailed design of processing on the compressed data in memory (Step 5), without considering whether the compressed data are already stored in memory (Step 2) or need to be read from disk to memory (Steps 3 and 4).

5 Detailed Design

The detailed design of POCLib is shown in Fig. 5, which consists of three parts: API design, library, and assistant tools. First, the framework provides APIs from both traversal operation [1], [2], [3] and random access mentioned in Section 4.2. Second, POCLib provides the function implementations in accordance with the APIs, and the data structures supporting these operations. Because we develop POCLib based on TADOC, we only introduce the common data structures for random accesses in Section 5.1 and the related operations, while remain the details to the TADOC traversal-related functions and data structures in [1], [2], [3]. Third, POCLib also includes compressor, decompressor, and profiling tool for analyzing text analytics operations.

Novelty. POCLib involves a series of novel techniques. First, our framework supports local graph walks (or partial

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

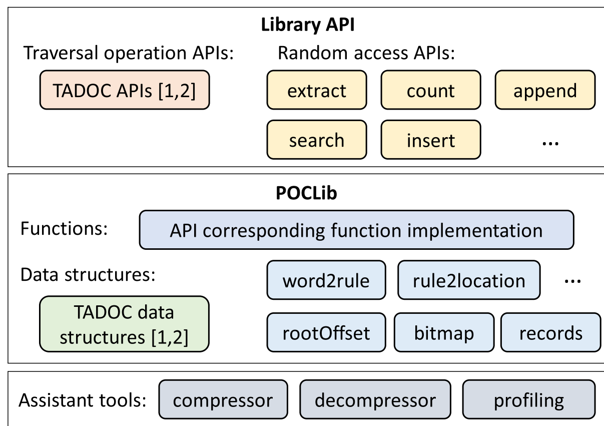


Fig. 5. Detailed design of POCLib.

traversals) starting from any place of interest in the DAG. This technique is essential for *extract* operation (Section 5.2). Second, our framework 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 (Sections 5.3 and 5.4). Third, our framework supports incremental dataset updates on the hierarchical compressed data. This technique makes efficient *insert* and *append* operations possible (Sections 5.5 and 5.6). We further consider graph *coarsening* as an optimization to save space cost (Section 5.8.3). These techniques are not independent of each other; they work synergistically to address the various complexities in all types of random accesses.

5.1 Common Data Structures

When designing the extra data structures required for each technique in POCLib, 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 5.2).
- *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 5.3).
- *rootOffset*. This data structure provides the offset of each element from the root rule (Section 5.5).
- *bitmap*.This data structure indicates whether or not an element in a rule has been changed (Section 5.5).
- *records*. This data structure stores the new content (Section 5.5).

These five data structures are designed to help TADOC with random accesses for near orthogonal POC in POCLib.

For quickly locating words given the 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).

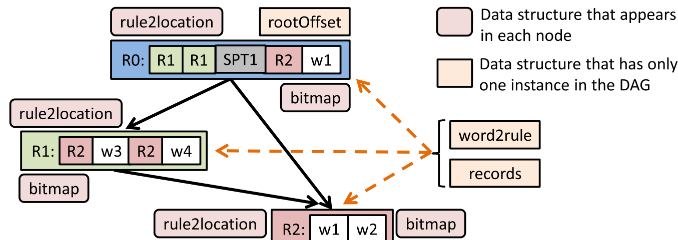


Fig. 6. Relationships between our new data structures.

For uni-directionality in TADOC storage format, 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.

For compressed data update of *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 Fig. 6. *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 5.6 provides more details.

Next, we explain in detail how our proposed techniques support each of the random access types in POCLib, and how to add a new operation in POCLib.

5.2 Extract

This operation extracts content directly from a compressed file. It is a basic operation required for reading data in 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. To demonstrate this challenge, we

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

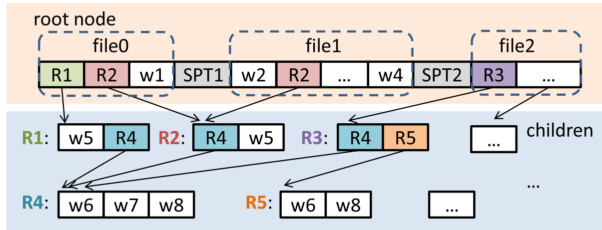


Fig. 7. An example of DAG representation for "w5 w6 w7 w8 w6 ...".

use the example shown in Fig. 7. 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.

Detailed Design of Our First Approach. We illustrate our pseudo-code of *extract* in Algorithm 1. The operation *extract* has three input parameters, file f, starting offset *start*, and the length of data to be extracted *len*. We first need to check whether the required content (f,*start*,*len*) is within the range, so we check the data structure *rule2location* of the root rule (line 3). If the required content does not exist, *extract* terminates (lines 4 to 5). We use an assistant data structure, *leftCount* (line 6) for calculating the distance to the required content; when its value is zero, it means that the starting offset has been found. The *ruleStartLoc* represents the offset of a rule. Because the root rule contains the information of different files, we need to traverse the DAG from a certain file offset (line 9). In the element traversal in the root, we can skip the elements that do not cover the range to avoid unnecessary data scanning if *leftCount* is larger than zero (lines 13 to 14, 22 to 23). If it is a word and this value is less than zero, we call a function *genOutput* to add the related content to the result (line 16); if we find a rule that covers the starting location, we call a function, *extractScan*, to recursively traverse the DAG and extract the required content.

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 Fig. 7, 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 Fig. 7. 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.

Algorithm 1. Extract len Bytes From $start$ in File f

1: **function** extract$f, start, len$
2: $end = start + len$
3: $find = checkRange(f, start, end, root)$
4: **if** $find == false$ **then**
5: return $wrong$
6: $leftCount = start - ruleStartLoc$
7: $rootStart = splitLocation[f]$
8: $rootEnd = splitLocation[f + 1]$
9: **for** each element i from $rootStart$ to $rootEnd$ **do**
10: **if** i is a word **then**
11: $checkUpdateExtract(f, start, len, i)$
12: $tmp = leftCount - wordLength[i]$
13: **if** $tmp > 0$ **then**
14: $leftCount = tmp$
15: **else**
16: $app = genOutput(i, output, leftCount, len)$
17: $len = len - app$
18: $leftCount = 0$
19: **else** ▷ i is a rule
20: $ruleLength = getLength(rule2location[i])$
21: $tmp = leftCount - ruleLength$
22: **if** $tmp > 0$ **then**
23: $leftCount = tmp$
24: **else**
25: $app = extractScan(i, output, leftCount, len)$
26: $len = len - app$
27: $leftCount = 0$
28: **if** $(leftCount == 0) and (len == 0)$ **then**
29: $break$
30: **return** $output$
31: **function** extractScan$i, output, leftCount, len$
32: $lenOriginal = len$
33: **for** each element j in rule i **do**
34: **if** j is a word **then**
35: $checkUpdateExtract(f, start, len, j)$
36: process j, similar to that in $extract()$
37: **else** ▷ j is a rule
38: $ruleLength = getLength(rule2location[j])$
39: $tmp = leftCount - ruleLength$
40: **if** $tmp > 0$ **then**
41: $leftCount = tmp$
42: **else**
43: $app = extractScan(i, output, leftCount, len)$
44: $len = len - app$
45: $leftCount = 0$
46: **if** $(leftCount == 0) and (len == 0)$ **then**
47: break
48: return $(lenOriginal - len)$

Let us examine the challenge of how to maintain pointers from children to parents. As Fig. 7 shows, a child such as R4

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

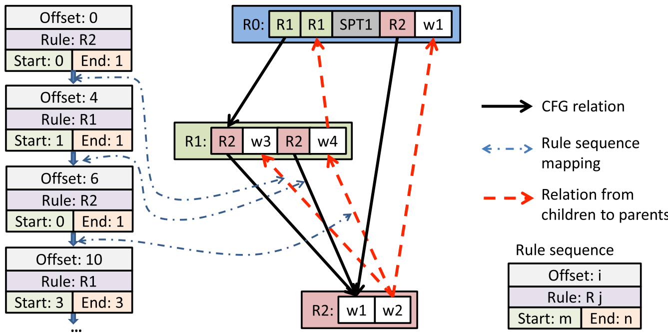


Fig. 8. Illustration of the *rule sequence* data structure for indexing. Assume the length of each word is two bytes.

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 Fig. 7, 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 Fig. 7, 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 Fig. 7, 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 Fig. 8. We use the DAG in Fig. 1c of Section 2 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 2 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 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 2. 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** $output$

In Section 7.5, we compare our first approach (Approach 1), the coarse-grained method, and our second approach (Approach 2), the fine-grained method of Algorithm 2 in detail.

5.3 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 Fig. 7, 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 5.1. 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 3. *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

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

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 Fig. 7, `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 Sections 5.5 and 5.6).

Algorithm 3. 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 our approach 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 Fig. 9. 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

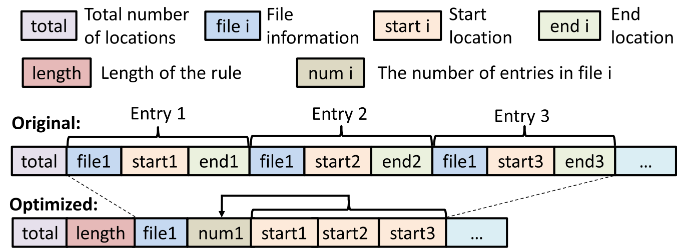


Fig. 9. Illustration of *rule2location* optimization.

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 5.8.3.

5.4 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 [1], [2], [3] 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 5.3, 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: $< word_i, set < (rule_a, freq_a), (rule_b, freq_b), \ldots > >$, 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 Fig. 9 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 Fig. 7, 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.

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

Detailed Design of Our Optimized Method. Algorithm 4 shows our optimized algorithm for *count*. The data structure *ruleFreq* stores the rule frequency in each file, and its format is $< rule_i, set < (file_a, freq_a), (file_b, freq_b), \ldots > >$, where $freq_j$ refers to the frequency of $rule_i$ in $file_j$. Because the *root* rule contains the file splitters (as shown in Fig. 7), 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 (Sections 5.5 and 5.6). In Section 7.5, we compare our first approach, the basic method (based on *search*), and our second approach, the optimized method (Algorithm 4) in detail.

Algorithm 4. 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
```

5.5 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 Fig. 7, 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 Fig. 7, `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".

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

(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".

5.6 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.

5.7 Sample

In this part, we show how POCLib supports a new operation, *sample*. This operation samples words directly from the compressed data, which is a basic operation for generating samples for a file. A possible design for *sample* is to randomly select a rule and return a word from the rule. However, this design is unsuitable because the probability of each rule is different. Furthermore, rules can belong to different files. Therefore, we abandon this design. Another method is to use the *ruleSequence* data structure (detailed in Section 5.2). We generate a random offset within the file first. Then, we locate the rule and return the word. Because we only return one word in *sample*, this process is much simpler than that in *extract*.

As our implementation makes no direct changes to the DAG, it ensures that other analytics, including the traversal operations [1], [2], [3], 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 7.4 evaluates the performance impact of the post-processing step on traversal operations.

5.8 Discussion

5.8.1 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.

5.8.2 Summary of Data Structures

Table 1 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 Sections 5.2 and 5.4, can be implemented using two different approaches, which use different data structures.

5.8.3 Space Considerations

As stated in Section 5.1, we introduce five additional data structures to support TADOC with orthogonal POC. For *word2rule*, we already presented its optimized format in Section 5.4. For *rule2location*, we have shown its optimization in Fig. 9 of Section 5.3. We have also illustrated the *records* data structure in Section 5.5. 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* [2]. 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

TABLE 1
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	
sample		DAG/dictionary	ruleSequence

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply.

structures. We analyze its effect on both space and performance in Section 7.5.

5.8.4 Limitation

POCLib supports text analytics that can be transformed into the interpretation of grammar rules. The limitation of POCLib is that currently, it only supports text analytics. We leave the support to other data types as our future work.

6 Implementation

We implement POCLib based on TADOC [2]. In POCLib, 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. *Sample* module returns a random word. 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 six modules, we also integrate a preprocessing stage to generate the necessary data structures, such as *word2rule*, *rule2-location*, *rootOffset*, *bitmap*, and *records*, for near orthogonal POC.

7 Evaluation

7.1 Methodology

The baseline method we compare to is Succinct [5]. 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 POCLib 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 “POCLib”. As ”POCLib” is based on TADOC, we keep the inputs the same as those to TADOC [2], 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*, *append*, and *sample*) on the preprocessed data.

We automatically generate the inputs of the six 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* and *sample* are similar.

TABLE 2
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

Datasets. Our evaluation uses five datasets that were used in previous studies [1], [2], [3], shown in Table 2. 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]. Datasets D and E are much smaller than datasets A, B, and C, so datasets D and E are used to evaluate performance on the single machine.

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 (datasets A, B, and C are too large on the single node machine). 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.

7.2 Performance

7.2.1 Large Datasets

Fig. 10 shows the throughput results (in terms of operations per second) for large datasets A, B, and C on the Spark cluster. In general, the six random access operations experience much higher throughput with our technique over Succinct. *Search* and *insert* have relatively low performance, *extract*, *count*, and *sample* 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

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply.

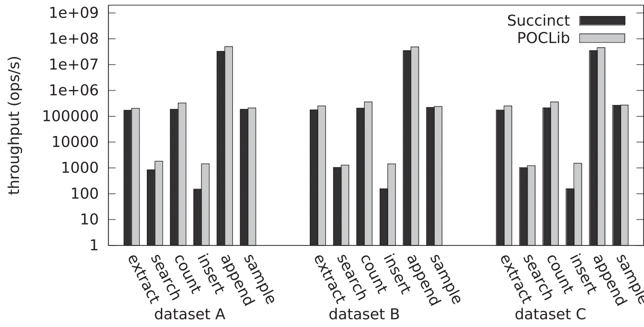


Fig. 10. Throughput for *extract*, *search*, *count*, *insert*, *append*, and *sample* on different datasets on the Spark cluster.

representation of Succinct and our hierarchical compressed representation. *Count*, *extract*, and *sample* do not have such overhead; *count* can obtain all the necessary information from *word2rule* and *rule2location*, and *extract* and *sample* concentrate 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.

Experiments show that our system consistently outperforms Succinct for the six operations on three datasets. For instance, POCLib achieves 234,764 *extract* operations per second on average, outperforming Succinct by 1.4×. POCLib achieves 1,443 *search* operations per second, outperforming Succinct by 1.5×. POCLib achieves 347,670 *count*, 1,462 *insert*, and 47,755,960 *append*, and 240,068 *sample* 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.

Fig. 11 shows the latency (in microseconds) of the six 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 5.6. Experiments show that Succinct exhibits better latency than POCLib, which relates to its simplicity and network fluctuation in the distributed environment. In contrast, the *search* and *insert* operations have relatively high latency, due to their complex interactions with the whole DAG. For the six operations, our system provides much lower latency than Succinct on most datasets: on average, POCLib reduces average operation

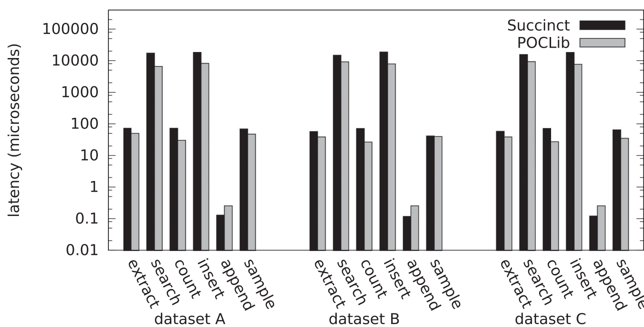


Fig. 11. Latency for *extract*, *search*, *count*, *insert*, *append*, and *sample* on different datasets on the Spark cluster.

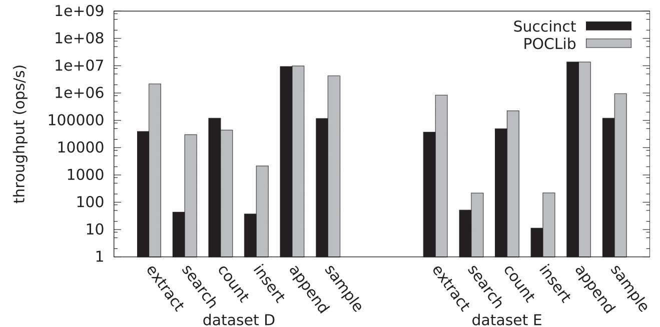


Fig. 12. Throughput for *extract*, *search*, *count*, *insert*, *append*, and *sample* on different datasets on the Single Node machine.

latency by 17.5 percent over Succinct (*append* drags down POCLib's average performance).

In summary, the data locality and data structures used in operations decide the performance. For example, *append* only needs to add data directly at the end of files, so it has the highest performance. In contrast, *search* relates to massive data accesses and thus has a relatively low performance. The throughput and latency of different datasets for the same operation are similar. The reason is that POCLib uses unified partitioning strategy, which makes the average content such as the number of rules for each RDD similar.

7.2.2 Small Datasets

Fig. 12 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.

The latency results for small datasets on the `Single Node` machine are shown in Fig. 13. The latency results for *extract*, *search*, *count*, *insert*, *append*, and *sample* are 1, 2,365, 14, 2,557, 0.1, and 0.7 microseconds on average. From Fig. 13, we can see that our method shows large performance benefits in most cases.

7.3 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 3. POCLib improves on TADOC [2] via the new data structures, as mentioned in Section 5.1, 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 disk because initially no insertion happens. Therefore, the only

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

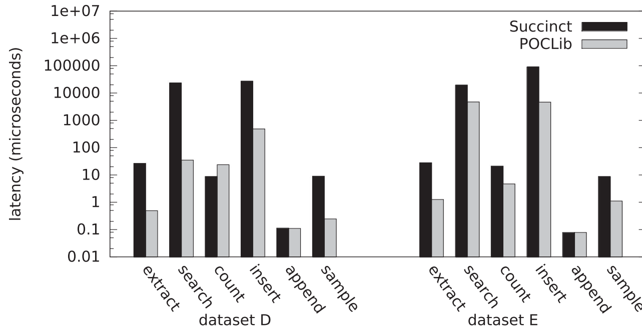


Fig. 13. Latency for *extract*, *search*, *count*, *insert*, *append*, and *sample* on different datasets on the Single Node machine.

data structures that incur disk storage cost are *rule2location*, *word2rule*, and *ruleFreq*, and their space breakdown in storage is shown in Table 4; *rule2location* occupies most of the space.

7.4 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 [1], [2], [3] 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 [2]. 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 the previous work [2], and 2) post-processing, which processes the new content in *records*. This means that, if no data are added into the dataset (no *insert* or *append* operations), POCLib preserves the performance of TADOC for traversal operations. If new data are added, POCLib requires additional post processing, which incurs time overhead. 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 5 reports the fraction of time spent on post-processing in each of the six traversal text analytics workloads. The ratio ranges from 3.7 to 30.4 percent, confirming that with our method, TADOC can now effectively handle data insertion with *insert* and *append* operations.

7.5 Tradeoff Between Performance and Space

7.5.1 Different Data Structures

The tradeoff between time and space is affected by the choices of data structures. In Section 4.2, we discuss two versions of *count* and *extract*. Table 6 provides a detailed analysis of time and memory consumption of each version. In our evaluation, our Approach 2 to *extract* (Algorithm 2 in Section 5.2) achieves an average of 9,756× throughput improvement over that of Approach 1 in Section 5.2). Our Approach 2 to *count* (second approach in Section 5.4) achieves an average of 70× throughput improvement over Approach 1 (the basic *count* version in Section 5.4). However, for *extract*, Approach 1 has smaller memory consumption; the reason is that Approach 2 generates *ruleSequence* during runtime, which consumes large memory space.

7.5.2 Coarsening

The tradeoff between time and space is also affected by coarsening. Coarsening is an optimization that makes each rule contain more elements, so that the total number of rules can be reduced, as discussed in Section 5.8.3. For illustration, we show the memory size of four data structures before and after coarsening for dataset A in Table 7. Accordingly, the sizes of *rule2location*, *ruleFreq*, and *ruleSequence* decrease after coarsening; the size of *word2rule* increases because each word is now contained in more rules.

Table 8 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 8 shows, the space savings from coarsening is over 62 percent 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 8 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

TABLE 3
Compression Ratios

Version	Dataset					
	A	B	C	D	E	AVG
Uncompressed	1.0	1.0	1.0	1.0	1.0	1.0
Succinct [5]	2.2	1.7	1.6	2.9	2.2	1.8
Original TADOC [2]	14.1	13.3	13.1	6.5	11.9	11.8
POCLib	4.9	3.5	3.5	2.6	5.0	3.9

TABLE 4
Space Breakdown for Different Data Structures (MB)

Data Structure	A	B	C	D	E
rule2location	4,707	13,427	26,815	122	192
word2rule	433	1,218	2,442	15	14
ruleFreq	27	76	151	65	2.1

TABLE 5
Fraction of Time Spent on Post-Processing

Application	Fraction of time on each dataset (%)					
	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

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

TABLE 6
Throughput and Memory Consumption Breakdown of Different Implementations of *Count* and *Extract*

Operation	Dataset	Throughput (ops/second)		Memory (MB)	
		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

implementation chooses to employ coarsening as coarsening provides a more desirable trade-off between space savings and speedup.

8 Related Work

To our knowledge, this work is the first to enable near orthogonal processing on compression for text analytics. We overcome the limitations of TADOC [1], [2], [3] 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, which can support advanced application scenarios [3], [23], [24], [25].

Sequitur is a well-known grammar-based compression algorithm [4], [26], [27]. It is first used for direct processing on compressed data by TADOC [1], [2], [3]. Besides text analytics, Sequitur is used for various other purposes, such as improving data reference locality [28], dynamic hot data stream prefetching [29], analyzing whole program paths [30], [31], finding loop patterns in program analysis [32], XML query processing [33], and comprehension of program traces [34].

Succinct [5] 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 [35] as basic compression structures, while our work extends a hierarchical compression method, Sequitur [4]. In terms of applications, Succinct is designed for database queries while our work is designed for general text analytics. Importantly, Succinct [5] 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 7 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 [36], [37].

Other prior works enable grammar compression over grammar-encoded text [38], [39], [40], [41], [42], [43], [44], [45]. Moreover, inverted index compression has also been fully studied [46], [47], [48], [49], [50], [51], [52]. Deduplication [53], [54], [55], [56], [57] is also a hot research topic. For example, Xia *et al.* [54] proposed a novel Content-Defined Chunking approach called FastCDC, which uses techniques of Gear-based rolling hash, normalized chunking, cut-points skipping, etc. FastCDC is extremely fast and has been used by lots of storage systems such as Ceph, rdedup, etc. for data deduplication. Different from these works, our work extends TADOC to orthogonal POC, and enables both data traversal operations and random accesses.

9 Conclusion

This paper presents a set of new solutions that enable efficient random accesses on hierarchically compressed data, significantly expanding the capability of text analytics on

TABLE 7
Memory Size of Different Data Structures Before and After Coarsening for Dataset A [22]

Data Structure	Before (MB)	After (MB)	Saving (%)
rule2location	23,535	5,938	74.8
word2rule	2,046	2,783	-36.0
ruleFreq	220	158	28.2
ruleSequence	72,729	29,341	59.7
Total	98,530	38,220	61.2

TABLE 8
Storage Savings With Coarsening and the Potential Speedup When Coarsening is Not Used

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

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

compressed data. It makes POCLib the first library that efficiently supports both traversal and random accesses directly on compressed texts. The new solutions consist of technical contributions in two parts. The first is a range of carefully designed indexing data structures. The design emphasizes reusability across analytics operations, and strikes a good balance between space cost and efficiency through selective on-the-fly indexing construction. The second is a set of algorithmic optimizations for the common random accesses to work efficiently on the compressed data. These optimizations help maximize the performance of data accesses by effectively leveraging the indexing data structures, incremental updates and recompression, and DAG coarsening. The technique makes the POCLib able to achieve 3.1× speedup over the state-of-the-art on random data accesses on compressed data, and at the same time, keeps its distinctive capability in supporting traversal operations efficiently. It makes the extended TADOC the first near orthogonal POC technique.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1004401, in part by the National Natural Science Foundation of China under Grants 61732014, U20A20226, and 61802412, in part by Beijing Natural Science Foundation under Grant 4202031, in part by the Beijing Academy of Artificial Intelligence (BAAI), and in part by the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCHA202007.

References

[1] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Zwift: A programming framework for high performance text analytics on compressed data," in *Proc. Int. Conf. Supercomputing*, 2018, pp. 195–206.
[2] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1522–1535, 2018.
[3] F. Zhang *et al.*, "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, pp. 163–188, 2021.
[4] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, vol. 7, pp. 67–82, 1997.
[5] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.
[6] B. Zhao and S. Vogel, "Adaptive parallel sentences mining from web bilingual news collection," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 745–748.
[7] A. B. Bepko, "Public availability or practical obscurity: The debate over public access to court records on the internet," *NYLS Law Rev.*, vol. 49, pp. 966–991, 2004.
[8] P. A. Winn, "Online court records: Balancing judicial accountability and privacy in an age of electronic information," *Washington Law Rev.*, vol. 79, 2004, Art. no. 307.
[9] S. Bao, J. Chen, L. C. En, R. Ma, and Z. Su, "Method and apparatus for enhancing webpage browsing: U.S. Patent," vol. 8, no. 577, p. 900, 2013.
[10] S. Lawrence and C. L. Giles, "Context and page analysis for improved Web search," *IEEE Internet Comput.*, vol. 2, no. 4, pp. 38–46, Jul./Aug. 1998.
[11] B. Zhang *et al.*, "The cloud is not enough: Saving IoT from the cloud," in *Proc. 7th USENIX Workshop Hot Top. Cloud Comput.*, 2015, pp. 1–7.
[12] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: Promise and potential," *Health Inf. Sci. Syst.*, vol. 2, 2014, Art. no. 3.
[13] R. H. Miller and I. Sim, "Physicians' use of electronic medical records: barriers and solutions," *Health Affairs*, vol. 23, no. 2, pp. 116–126, 2004.
[14] P. Jalan, A. K. Jain, and S. Roy, "Identifying hierarchical structures in sequences on GPU," in *Proc. Trustcom/BigDataSE/ISPA*, 2015, pp. 27–36.
[15] M. Zaharia *et al.*, "Spark: Cluster computing with working sets [J]," *HotCloud*, vol. 10, no. 10–10, p. 95, 2010.
[16] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 841–850.
[17] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM J. Comput.*, vol. 35, no. 2, pp. 378–407, 2005.
[18] Gzip, 2019. [Online]. Available: https://www.gzip.org/
[19] Wikipedia HTML data dumps, 2017. [Online]. Available: https://dumps.wikimedia.org/enwiki/
[20] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[21] Amazon EC2, 2019. [Online]. Available: https://aws.amazon.com/ec2/
[22] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1069–1080.
[23] F. Zhang *et al.*, "G-TADOC: Enabling efficient GPU-based text analytics without decompression," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 1679–1690.
[24] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 905–918, Mar. 2017.
[25] J. Zhang, B. Hao, B. Chen, C. Li, H. Chen, and J. Sun, "Hierarchical reinforcement learning for course recommendation in MOOCs," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 435–442.
[26] C. G. Nevill-Manning, "Inferring sequential structure," Ph.D. dissertation, Univ. Waikato, Hamilton, New Zealand, 1996.
[27] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Proc. Data Compression Conf.*, 1997, pp. 3–11.
[28] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2001, pp. 191–202.
[29] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2002, pp. 199–209.
[30] J. R. Larus, "Whole program paths," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 1999, pp. 259–269.
[31] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proc. Int. Conf. Softw. Eng.*, 2003, pp. 308–318.
[32] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2005, pp. 135–146.
[33] Y. Lin, Y. Zhang, Q. Li, and J. Yang, "Supporting efficient query processing on compressed XML files," in *Proc. of the 2005 ACM Symp. on Applied Comput.*, 2005.
[34] N. Walkinshaw, S. Afshan, and P. McMinn, "Using compression algorithms to support the comprehension of program traces," in *Proc. 8th Int. Workshop Dyn. Anal.*, 2010.
[35] G. Navarro, *Compact Data Structures: A Practical Approach. Cambridge Univ. Press*, 2016.
[36] A. Khandelwal, R. Agarwal, and I. Stoica, "BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores," in *USENIX Symp. Netw. Syst. Des. Implementation*, 2016.
[37] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica, "ZipG: A Memory-efficient Graph Store for Interactive Queries," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1149–1164.
[38] W. Rytter, "Grammar compression, LZ-encodings, and string algorithms with implicit input," in *Proc. Int. Colloq. Automata, Lang., Program.*, 2004, pp. 15–27.
[39] M. Charikar *et al.*, "The smallest grammar problem," *IEEE Trans. Inf. Theory*, vol. 51, no. 7, pp. 2554–2576, Jul. 2005.
[40] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi, "A faster grammar-based self-index," in *Proc. Int. Conf. Lang. Automata Theory Appl.*, 2012, pp. 240–251.

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply

[41] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann, "Random access to grammar-compressed strings and trees," *SIAM J. Comput.*, vol. 44, pp. 513–539, 2015.

[42] P. Bille, A. R. Christiansen, P. H. Cording, and I. L. Gørtz, "Finger search in grammar-compressed strings," 2015, *arXiv:1507.02853*.

[43] N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. R. Paramá, "GraCT: A grammar-based compressed index for trajectory data," *Inf. Sci.*, vol. 483, pp. 106–135, 2019.

[44] M. Ganardi, A. Jeż, and M. Lohrey, "Balancing straight-line programs," in *Proc. IEEE 60th Annu. Symp. Found. Comput. Sci. (FOCS)*, 2019, pp. 1169–1183.

[45] Y. Takabatake and H. Sakamoto, "A space-optimal grammar compression," in *Proc. 25th Annu. Eur. Symp. Algorithms*, 2017, pp. 67:1–67:15.

[46] M. Petri and A. Moffat, "Compact inverted index storage using general-purpose compression libraries," *Softw.: Pract. Experience*, vol. 48, pp. 974–982, 2018.

[47] A. Moffat and M. Petri, "Index compression using byte-aligned ANS coding and two-dimensional contexts," in *Proc. 11th ACM Int. Conf. Web Search Data Mining*, 2018, pp. 405–413.

[48] G. E. Pibiri, M. Petri, and A. Moffat, "Fast dictionary-based compression for inverted indexes," in *Proc. 12th ACM Int. Conf. Web Search Data Mining*, 2019, pp. 6–14.

[49] G. E. Pibiri and R. Venturini, "Techniques for inverted index compression," 2019, *arXiv:1908.10598*.

[50] G. E. Pibiri, R. Perego, and R. Venturini, "Compressed indexes for fast search of semantic data," *IEEE Trans. Knowl. Data Eng.*, early access, Jan. 14, 2020, doi: 10.1109/TKDE.2020.2966609.

[51] H. Oosterhuis, J. S. Culpepper, and M. de Rijke, "The potential of learned index structures for index compression," in *Proc. 23rd Australas. Document Comput. Symp.*, 2018, pp. 1–4.

[52] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel, "Compressing inverted indexes with recursive graph bisection: A reproducibility study," in *Proc. Eur. Conf. Inf. Retrieval*, 2019, pp. 339–352.

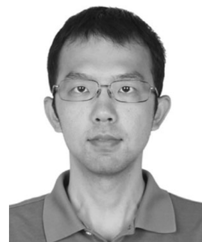
[53] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 26–28.

[54] W. Xia *et al.*, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2016, pp. 101–114.

[55] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[56] X. Zou, *et al.*, "The Dilemma between Deduplication and Locality: Can Both be Achieved?," in *Proc. 19th Conf. File Storage Technologies*, 2021, pp. 171–185.

[57] W. Xia, "The design of fast content-defined chunking for data deduplication based storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2017–2031, Sep. 2020.



Feng Zhang received the bachelor's degree from Xidian University in 2012 and the PhD degree in computer science from Tsinghua University in 2017. He is currently an associate professor with DEKE Lab and the School of Information, Renmin University of China. His main research interests include database systems, and parallel and distributed systems.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China in 2003 and the PhD degree in computer science from Tsinghua University in 2010. He is currently an associate professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis, and modeling of parallel applications.

Xipeng Shen received the PhD degree in computer science from the University of Rochester in 2006. He is currently a professor of computer science with the North Carolina State University. His research interests include programming systems and machine learning. He was the recipient of the DOE Early Career Award, the NSF CAREER Award, the Google Faculty Research Award, and the IBM CAS Faculty fellow Award. He is a distinguished member and a distinguished speaker of ACM.

Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and the MS and PhD degrees in electrical and computer engineering from the University of Texas at Austin. He is currently a professor of computer science with ETH Zurich. He is also a faculty member with Carnegie Mellon University, where he previously held the Strecker Early Career Professorship. His current research interests include computer architecture, systems, hardware security, and bioinformatics.

Xiaoyong Du received the BS degree from Hangzhou University, Zhengjiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

uthorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on October 15,2021 at 09:38:28 UTC from IEEE Xplore. Restrictions apply