**Noname manuscript No.**
(will be inserted by the editor)

# TADOC: Text Analytics Directly on Compression

**Feng Zhang**[1] · **Jidong Zhai**[2] · **Xipeng Shen**[3] · **Dalin Wang**[1] · **Zheng Chen**[1] · **Onur Mutlu**[4] · **Wenguang Chen**[2] · **Xiaoyong Du**[1]



Received: date / Accepted: date

**Abstract** This article provides a comprehensive description of Text Analytics Directly on Compression (TADOC), which enables direct document analytics on compressed textual data. The article explains the concept of TADOC and the challenges to its effective realizations. Additionally, a series of guidelines and technical solutions that effectively address those challenges, including the adoption of a hierarchical compression method and a set of novel algorithms and data structure designs, are presented. Experiments on six data analytics tasks of various complexities show that TADOC can save 90.8% storage space and 87.9% memory usage, while halving data processing times.

Feng Zhang
E-mail: fengzhang@ruc.edu.cn

Jidong Zhai
E-mail: zhaijidong@tsinghua.edu.cn

Xipeng Shen
E-mail: xshen5@ncsu.edu

Dalin Wang
E-mail: sxwangdalin@ruc.edu.cn

Zheng Chen
E-mail: 2016202201@ruc.edu.cn

Onur Mutlu
E-mail: onur.mutlu@inf.ethz.ch

Wenguang Chen
E-mail: cwg@tsinghua.edu.cn

Xiaoyong Du
E-mail: duyong@ruc.edu.cn

[1]DEKE Lab, School of Information, Renmin University of China, China

[2]Department of Computer Science and Technology, Tsinghua University, China

[3]Computer Science Department, North Carolina State University, USA

[4]Department of Computer Science, ETH Zürich, Switzerland

## 1 Introduction

*Document analytics* refers to data analytics tasks that derive statistics, patterns, insights or knowledge from textual documents (e.g., system log files, emails, corpus). It is important for many applications, from web search to system diagnosis, security, and so on. Document analytics applications are time-consuming, especially as the data they process keep growing rapidly. At the same time, they often need a large amount of space, both in storage and memory.

A common approach to mitigating the space concern is data compression. Although it often reduces the storage usage by several factors, compression does *not* alleviate, but actually worsens, the time concern. In current document analytics frameworks, compressed documents have to be decompressed before being processed. The decompression step lengthens the end-to-end processing time.

This work investigates the feasibility of efficient data analytics on compressed data *without* decompressing it. Its motivation is two-fold. First, it could avoid the decompression time. Second, more importantly, it could save some processing. Space savings by compression fundamentally stems from repetitions in the data. If the analytics algorithms could leverage the repetitions that the compression algorithm already uncovers, it could avoid unnecessary repeated processing, and hence shorten the processing time significantly. Compression takes time. But many datasets (e.g., government document archives, electronic book collections, historical Wikipedia datasets [5]) are used for various analytics tasks by many users

repeatedly. For them, the compression time is well justified by the repeated usage of the compression results.

This article presents Text Analytics Directly on Compression (TADOC), which enables direct document analytics on compressed textual data. We base TADOC on a specific compression algorithm named *Sequitur* [65] for the hierarchical structure of its compression results (Section 2).

We introduce the concept of *compression-based direct processing*, and analyze its challenges (Section 3). Through studies on a set of core algorithms used in document analytics, we discover a set of solutions and insights on tackling those challenges. These insights range from algorithm designs to data structure selections, scalable implementations, and adaptations to various problems and datasets. We draw on several common document analytics problems to explain our insights, and provide the first set of essential guidelines and techniques for effective compression-based document analytics (Section 4).

Our work yields an immediately-usable artifact, the `CompressDirect` library, which offers a set of modules to ease the application of our guidelines. Our library provides implementations of six algorithms frequently used in document analytics, in sequential, parallel, and distributed versions, which can be directly plugged into existing applications to generate immediate benefits.

We further discuss how TADOC and its associated CompressDirect library can be effectively applied to real-world data analytics applications. We demonstrate the process on four applications, *word co-occurrence* [58, 72], *term frequency-inverse document frequency* [41], *word2vec* [79,3], and *latent Dirichlet allocation* (LDA) [12]. Our evaluation validates the efficacy of our proposed techniques in saving both space and time on six analytics kernels. For six common analytics kernels, compared to their default implementation on uncompressed datasets, TADOC reduces storage usage by 90.8% and memory usage by 87.9%, and at the same time, speeds up the analytics by 1.6X for sequential runs, and by 2.2X for Spark-based distributed runs. On four real-world applications, TADOC reduces storage usage by 92.4% and memory usage by 26.1%, and yields 1.2X speedup over the original applications, on average.

A prior work, Succinct [8], offers a way to enable efficient queries on compressed data. This work complements it by making complex document analytics on compressed data efficiently. Data deduplication [61] saves storage space, but does *not* save repeated processing of the data. Our preliminary work has been presented in [97].

Overall, this work makes the following contributions:

- It presents an effective method for enabling high performance complex document analytics directly on compressed data, and realizes the method on the *Sequitur* compression algorithm.
- It unveils the challenges of performing *compression-based document analytics* and offers a set of solutions, insights, and guidelines.
- It validates the efficacy of the proposed techniques, demonstrates their significant benefits in both space and time savings, and offers a library for supporting common operations in document analytics.
- It demonstrates that TADOC can be effectively applied to real-world document analytics applications, bringing 92.4% storage space savings and 1.2X performance improvement.

## 2 Premises and Background

Operating directly on grammar-compressed data is an active research area in recent years [62,85], which will be discussed in Section 8. In this section, we present the premises and background of Sequitur compression algorithm and typical document analytics.

### 2.1 Sequitur Algorithm

There are many compression algorithms for documents, such as LZ77 [100], suffix array [62], and their variants. Our study focuses on Sequitur [65] since its compression results are a natural fit for direct processing.

Sequitur is a recursive algorithm that infers a hierarchical structure from a sequence of discrete symbols. For a given sequence of symbols, it derives a context-free grammar (CFG), with each rule in the CFG reducing a repeatedly appearing string into a single rule ID. By replacing the original string with the rule ID in the CFG, Sequitur makes it output CFG more compact than the original dataset.

Figure 1 illustrates Sequitur compression results. Figure 1 (a) shows the original input, and Figure 1 (b) shows the output of Sequitur in a CFG form. The CFG uncovers both the repetitions in the input string and the hierarchical structure. It uses R0 to represent the entire string, which consists of substrings represented by R1 and R2. The two instances of R1 in R0 reflect the repetition of "a b c a b d" in the input string, while the two instances of R2 in R1 reflect the repetition of "a b" in the substring of R1. The output of Sequitur is often visualized with a directed acyclic graph (DAG), as Figure 1 (c) shows. The edges indicate the hierarchical relations among the rules.

Dictionary encoding is often used to represent each word with a unique non-negative integer. A dictionary stores the mapping between integers and words. We represent each rule ID with a unique integer greater than $N$, where $N$ is the total number of unique words contained in the dataset. Figure 1 (d) gives the numerical

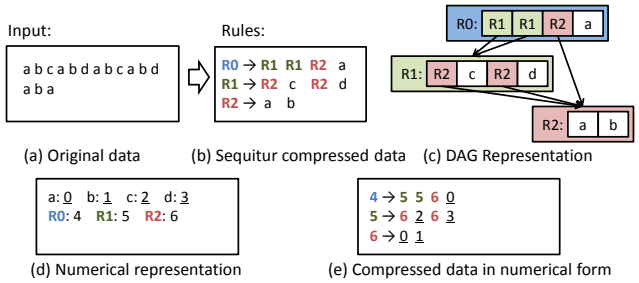


Fig. 1: A compression example with Sequitur.

representations of the words and rules in Figure 1 (a,b), while Figure 1 (e) shows the CFG in numerical form.

Sequitur provides compression ratios similar to those of other popular algorithms (e.g., Gzip) [63]. Its compression process is relatively slow, but our technique is designed for datasets that are *repeatedly* used by many users. For them, compression time is *not* a main concern as the compression results can be used many times by different users for various tasks. Such datasets are common, ranging from book collections to historical Wikipedia pages [5], government document archives, archived collections (e.g., of a law firm), historical news collections, and so on.

Sequitur has several properties that make it appealing for our use. First, the CFG structure in its results makes it easy to find repetitions in input strings. Second, its output consists of the direct (sequences of) input symbols rather than other indirect coding of the input (e.g., *distance* used in LZ77 [100] and suffix array [62]). These properties make Sequitur a good fit for materializing the idea of compression-based direct processing.

### 2.2 Typical Document Analytics

Before presenting the proposed technique, we first describe three commonly-performed document analytics tasks. They each feature different challenges that are typical to many document analytics, offering examples we use in later sections for discussion.

**Word Count** *Word count* [13,71,9] is a basic algorithm in document analytics, which is widely used in applications like document classification, clustering, and theme identification. It counts the total appearances of every word in a given dataset, which may consist of a number of files.

– Input: {file1, file2, file3, file4, file5, ...}
– Output: <word1, count1>, <word2, count2>, ...

**Inverted Index** *Inverted index* [9] builds word-to-file index for a document dataset. It is worth noting that in some implementations of *inverted index*, some extra operations are involved (e.g., getting and storing the term frequency) during the construction of the word-to-file index [101]. The implementation in our study is pure in computing only word-to-file index as the purpose is to demonstrate how TADOC can be used for various tasks. (We have a separate benchmark, *term vector*, for computing term frequencies.)

– Input: {file1, file2, file3, file4, file5, ...}
– Output: <word1, <file1>>, <word2, <file13>>, ...

**Sequence Count** *Sequence count* [9,94,49] counts the number of appearances of every $l$-word sequence in each file, where $l$ is an integer greater than 1. In this work, we use $l$=3 as an example. *Sequence count* is very useful in semantic, expression, and sequence analysis. Compared to *word count* and *inverted index*, *sequence count* not only distinguishes between different files, but also discerns the order of consecutive words, which poses more challenges for processing (Section 3).

– Input: {file1, file2, file3, file4, file5, ...}
– Output: <word1_word2_word3, file1, count1>, ...

## 3 TADOC and its Challenges

In this section, we present the concept of TADOC, including its basic algorithms and the challenges for materializing it effectively.

### 3.1 TADOC

TADOC is a technique that supports various text analytics tasks directly on compressed data without decompression. The compression is not task-specific. For example, we compress text files using TADOC, and the compressed data can be used directly to support text analytics tasks such as *word count* and *inverted index*. The basic concept of compression-based document analytics is to leverage the compression results for *direct processing* while avoiding unnecessary repeated processing of repeated content in the original data.

The results from Sequitur make this basic idea easy to materialize. Consider a task for counting word frequencies in input documents. We can do it directly on the DAG from Sequitur using a postorder (children before parents) traversal, as Figure 2 shows. After the DAG is loaded into memory, the traversal starts. At each node, it counts the frequency of each word that the node directly contains and calculates the frequencies of other words it indirectly contains—in its children nodes. For instance, when node R1 in Figure 2 is processed, direct appearances of "c" and "d" on its right-hand-side (*rhs*) are counted, while, the frequencies of words "a" and "b" are calculated by multiplying their frequencies in R2 by two—the number of times R2 appears on its *rhs*. When the traversal reaches the root R0, the algorithm produces the final answer.

Because the processing leverages the compression results, it naturally avoids repeated processing of repeated content. The repeated content needs to be counted

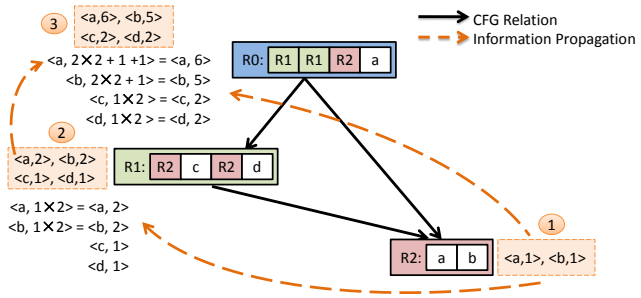


Fig. 2: A DAG from Sequitur for "a b c a b d a b c a b d a b a", and a postorder traversal of the DAG for counting word frequencies.

only once. For instance, even though the substring "a b" (R2) appears five times in the original string, the processing only counts its frequency once. It is counted when the algorithm processes R2; the results are reused for the two appearances of R2 when R1 is processed; similarly, R1's results (and also R2's) are reused when R0 is processed.

The procedure of postorder traversal for word counting is described in Algorithm 1, `wordcount-V1`. The algorithm associates with each rule a local table *locTbl*. During the data loading time (line 2 in Algorithm 1), when reading in each rule in the CFG, the algorithm records in the *locTbl* of that rule the frequency of each rule and each word appearing on the right-hand side of that rule.

**Algorithm 1** `wordcount-V1`

```
1:  init()                                  ▷ nullify elements of Boolean array done
2:  G=LoadMergeGraph(I)                     ▷
    Load compressed dataset I, build the graph with edges
    between two nodes merged into one; each node has a local
    table locTbl, initialized with the numbers of each subrule
    and each word's appearances on the right-hand side of
    the rule represented by that node.
3:  countWords(G.root)
4:  procedure countWords(id)                ▷ id: a rule ID
5:      for each subRule i in rule id do
6:          if !done[i] then
7:              countWords(i)
8:          end if
9:      end for
10:     for each subRule i in rule id do
11:         n=rule[id].locTbl[i]            ▷ freq of i in rule id
12:         for each word w in the locTbl of rule i do
13:             rule[id].locTbl[w] += n * rule[i].locTbl[w]   ▷
    fold the counts into the parent node
14:         end for
15:     end for
16:     done[id] = true
17: end procedure
```

To analyze the complexity of Algorithm 1, we first define four concepts in the DAG.

*Reachable*: In the DAG, if a path exists from node $r_i$ to node $r_j$, then we say node $r_j$ is reachable from $r_i$.

*Reachable node set*: In the DAG, all the reachable nodes from node $r_i$ form a reachable node set of $r_i$.

*Reachable edge set*: In the DAG, the out edges of all the nodes in node $r_i$'s reachable node set form the reachable edge set of node $r_i$.

*Reverse reachable node set*: After all the edges in the DAG are reversed, all the reachable nodes from node $r_i$ form a reverse reachable node set of node $r_i$.

*Reverse reachable edge set*: After all the edges in the DAG are reversed, the out edges of all the nodes in node $r_i$'s reachable node set form the reachable edge set of node $r_i$.

The time complexity of Algorithm 1 is dominated by two parts: 1) building a local table in each rule (lines 1-2), and 2) accumulating the word counts from the local table to the root (note that all the reverse reachable edges need to be traversed). For the first part, all the elements in each rule need to be analyzed, so the complexity involved in building the local table is $n_e * k$, where $n_e$ is the average number of elements for the nodes and $k$ is the number of nodes in the DAG. For the second part, assume the number of elements in the local word table of node $i$ is $n_i$ and the number of edges in the reverse reachable edge set of the node $i$ is $e_i$, then the complexity in accumulating the word counts from the node $i$ to the root is $n_i * e_i$. Based on the analysis, the time complexity of Algorithm 1 is $O(n_e * k + \sum_0^{k-1}(n_i * e_i))$. The space complexity of Algorithm 1 is $O(s + g + k * l)$, where $s$ is the size of the DAG, $g$ is the global table size, $k$ is the number of nodes in the DAG, and $l$ is the average size of the local tables.

The example illustrates the essence of the general algorithm of our TADOC method:

> Let G be the graph representing Sequitur compression results.
> Conduct a traversal of G, during which, at each node, do the following:
> (1) Local Processing:
> Process local info;
> (2) Data Integration:
> Integrate the local info with results passed to this node during the traversal;
> (3) Data Propagation:
> Pass the integrated info to other nodes while continuing the traversal.

We name the three main operations *local processing*, *data integration*, and *data propagation* respectively. Document analytics is converted into a graph traversal process. Such a traversal process leverages the structure of the input documents captured by Sequitur, and embodies information reuse to avoid repeated processing of repeated content.

In terms of application scope, TADOC is designed for document analytics applications that can be expressed as DAG traversal-based problems on the compressed datasets, where the datasets do *not* change frequently. Such applications would fit and benefit most from our approach.

### 3.2 Challenges

Effectively materializing the concept of TADOC faces a number of challenges. As Figure 3 shows, these challenges center around the tension between reuse of results across nodes and the overheads in saving and propagating results. Reuse saves repeated processing of repeated content, but at the same time, requires the computation results to be saved in memory and propagated throughout the graph. The key to effective TADOC is to maximize the reuse while minimizing the overhead.

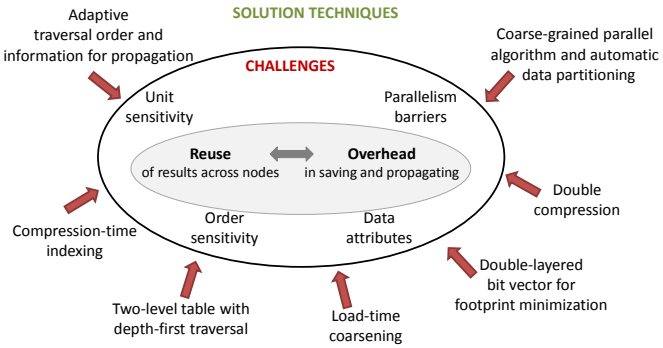


Fig. 3: Overview of the challenges and solutions.

Our problem is complicated by the complexities imposed by the various analytics problems, the large and various datasets, and the demand for scalable performance. We summarize the most common challenges as follows:

- *Unit sensitivity.* *Word count* regards the entire input dataset as a single bag of words. Yet, in many other document analytics tasks, the input dataset is regarded as a collection of some units (e.g., files). For instance, *inverted index* and *sequence count* try to get some statistics in each file. The default Sequitur compression does *not* discern among files. How to support unit sensitivity is a question to be answered.
- *Order sensitivity.* The way that results are propagated in the example of *word count* in Figure 2 neglects the appearance order of words in the input documents. A challenge is how to accommodate the order for applications (e.g., *sequence count*) that are sensitive to the order. This is especially tricky when a sequence of words span across the boundaries of multiple nodes (e.g., the ending substring "a b a" in Figure 1 spans across nodes R2 and R0).
- *Data attributes.* The attributes of input datasets, such as the number of files, the sizes of files, and the number of unique words, may sometimes substantially affect the overhead and benefits of a particular design for TADOC. For instance, when solving *inverted index*, one method is to propagate through the graph the list of files in which a word appears. This approach could be effective if there are a modest number of files, but would incur large propagation overheads otherwise, since the list to propagate could get very large. Thus, datasets with different properties could demand a different design in what to propagate and the overall traversal algorithm.
- *Parallelism barriers.* For large datasets, parallel or distributed processing is essential for performance. However, TADOC introduces some dependencies that form barriers. In Figure 2, for instance, because nodes R1 and R0 require results from the processing of R2, it is difficult for them to be processed concurrently with R2.

A naive solution to all these challenges is to decompress data before processing. However, doing so loses most benefits of compression. We next present our novel solutions to the challenges.

## 4 Guidelines and Techniques

This section presents our guidelines, techniques, and software modules for easing programmers' jobs in implementing efficient TADOC.

### 4.1 Solution Overview

The part outside the challenge circle in Figure 3 gives an overview of the solutions to the challenges. Because of the close interplay between various challenges, each of our solution techniques simultaneously relates with multiple challenges. They all contribute to our central goal: maximizing reuse while minimizing overhead.

The first solution technique is about the design of the graph traversal algorithm, emphasizing the selection of the traversal order and the information to propagate to adapt to different problems and datasets (Section 4.2). The second is about data structure design, which is especially useful for addressing unit sensitivity (Section 4.2). The third is on overcoming the parallelism barriers through coarse-grained parallel algorithm design and automatic data partition (Section 4.3). The fourth addresses order sensitivity (Section 4.4). The other three are some general optimizations to be applied at compression time and graph loading time, useful for both reducing the processing overhead and maximizing the compression ratio (Section 4.5). For these techniques, we have developed some software modules to assist programmers in using the techniques.

In the rest of this section, we describe each of the techniques along with the corresponding modules.

### 4.2 Adaptive Traversal Order

The first of the key insights we learned through our explorations is that graph traversal order significantly affects the efficiency of TADOC. Its influence is coupled with the information that the algorithm propagates through the graph during the processing. The appropriate traversal order choice depends on the characteristics of both the problems and the datasets.

In this part of paper, we first draw on *word count* and *inverted index* as two examples to explain this insight, and then present the derived guideline and a corresponding software module to assist developers. Through the way, we will also explain some basic operations needed to handle unit sensitivity of document analytics.

*An Alternative Algorithm for Word Count*

Figure 2 has already shown how *word count* can be done through a postorder traversal of the CFG. Doing so can yield a decent speedup by saving computation (e.g., 1.4X on a 2.1GB *Wikipedia* dataset [5], which is dataset E in Section 7.1). However, we find that if we change the traversal to preorder (parents before children), the speedups can almost double.

Figure 4 illustrates this alternative design. It consists of two main steps. The first step calculates the total frequency with which each rule appears in the entire dataset (① to ③ in Figure 4). This step is done through a preorder traversal of the graph: Parent nodes pass their total frequencies to their children, from which the children nodes can calculate their total frequencies. Let $f(r)$ be the computed frequency of rule $r$. With this step done, the second step (④ in Figure 4) just needs to enumerate all the rules once. No graph traversal is needed. When it processes rule $r$, it calculates $f_r(w) = c_r(w) * f(r)$, where, $c_r(w)$ is the directly observable frequency of the word $w$ on the right-hand side of rule $r$ (i.e., without considering the subrules of $r$), and $f_r(w)$ is the total directly observable frequency of $w$ due to all occurrences of rule $r$. Let $f(w)$ be the frequency of word $w$. The algorithm adds $f_r(w)$ into $f(w)$, the accumulated frequency of $w$. So when the enumeration of all rules is done, the total count of every word is produced. For example, $f_0(a)$ is 1, $f_1(a)$ is 0, and $f_2(a)$ is 5, so $f(a)$ is 6.

Based on the idea, we implement `wordcount-V2`, shown in Algorithm 2. It consists of two main steps. The first step (lines 5 to 14) calculates the frequencies of each rule, and the second step (lines 16 to 20) goes through the words on the right-hand side of each rule to obtain the accumulated total frequency of each word. The first step propagates the *fq* of each node to its children in a preorder (parents before children) traversal of the graph. Note that a node's *fq* should be propagated to its children only after its *fq* is fully ready—that is, all its parents' *fq*s have been folded into its *fq*. Algorithm 2 uses a queue *workQue* to ensure that: a rule is enqueued only when its *fq* has been updated $k$ times, where $k$ is the number of its parents (line 10 in Algorithm 2). Compared to `wordcount-V1`, this version needs to propagate only an integer *fq* between nodes, significantly reducing the runtime overhead.

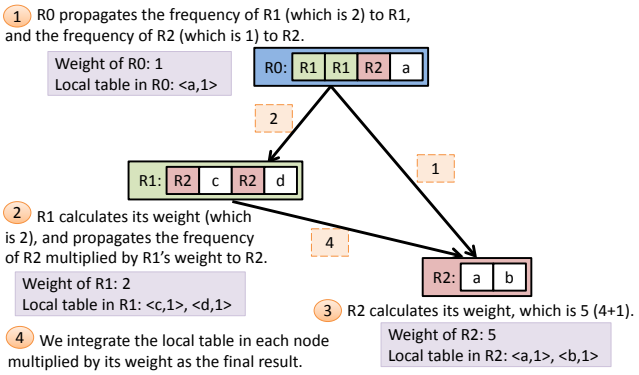


Fig. 4: Illustration of a preorder traversal algorithm for *word count* on the same string as Figure 2 deals with. The weights on an edge between two nodes (say, $node_x$ to $node_y$) indicate the total appearances of $node_y$'s rule due to all appearances of $node_x$'s rule.

**Algorithm 2** `wordcount-V2`

```
1:  G = LoadMergeGraph(I)                          ▷ same as in Algorithm 1 including setting up locTbl for each rule; additionally, each rule has a field fq set to 0, updates set to 0
2:  rule[G.root].fq = 1
3:  init workQue to empty
4:  workQue.enque(G.root)
5:  while !workQue.empty() do                      ▷ Calculate frequencies
6:      head = workQue.deque()
7:      for each subRule i in rule head do
8:          rule[i].fq+ = rule[head].fq * rule[head].locTbl[i]
9:          rule[i].updates++
10:         if rule[i].updates == rule[i].inEdges then   ▷ Done with updating its fq, ready to be propagated to its children
11:             workQue.enque(i)
12:         end if
13:     end for
14: end while
15: init(wordCounts)                               ▷ all zeros
16: for each rule i do                             ▷ accumulate word's frequency
17:     for each word w in rule[i] do
18:         wordCounts[w] += rule[i].locTbl[w] * rule[i].fq
19:     end for
20: end for
```

The time complexity of Algorithm 2 is dominated by two parts: 1) building a local table in each rule, and 2) calculating the accumulated frequency for each rule. The first part is the same as the first part in the complexity analysis of Algorithm 1. For the second part, the rule frequency is transmitted from the root to the children, cumulatively; each edge is passed only once, so the complexity of this part is the number of edges $d_G$. Therefore, the time complexity of Algorithm 2 is $O(n_e * k + d_G)$, which is much simpler than the complexity of Algorithm 1. The space complexity of Algorithm 2 is also $O(s + g + k * l)$, where $s$ is the size of the DAG, $g$

is the global table size, $k$ is the number of nodes in the DAG, and $l$ is the average size of the local tables.

This alternative algorithm achieves a much larger speedup (2.0X versus 1.4X on dataset E of *Wikipedia* in Section 7.1) than the postorder algorithm does. The reason is that it needs to propagate only an integer—the node's frequency—from a node to its children, while the postorder algorithm in Figure 2 needs to propagate the frequencies of all the words the node and the node's successors contain. This example illustrates the importance of traversal order and the information to propagate for the efficiency of TADOC.

*Traversals for Inverted Index*

The appropriate traversal order depends on both the analytic tasks and datasets. We illustrate this point by describing two alternative traversal algorithms designed for *inverted index*.

Recall that the goal of *inverted index* is to build a mapping from each word to the list of files in which it appears. Before we discuss the different traversal orders, we note that the objective of this analytics task requires discerning one file from another. Therefore, in the Sequitur compression results, file boundaries should be marked. To do so, we introduce a preprocessing step, which inserts some special markers at file boundaries in the original input dataset. As these markers are all distinctive and differ from the original data, in the Sequitur compressed data, they become part of the root rule, separating the different files, as the "spt1" and "spt2" in Figure 5 illustrate. (This usage of special markers offers a general way to handle unit sensitivity.)

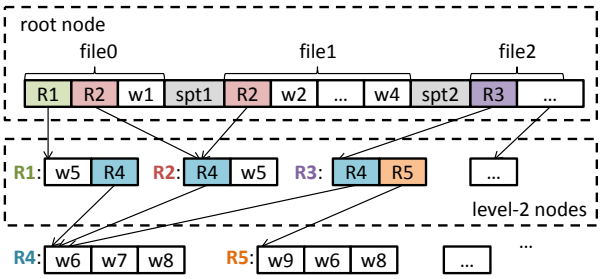


Fig. 5: Sequitur compression result with file separators ("spt1" and "spt2" are two file separators).

We next explain both preorder and postorder designs for *inverted index*. In the *preorder* design, the first step propagates the set of the IDs of the files that contain the string represented by the node, instead of the frequencies of rules. For instance, in Figure 5, the *fileSet* of rule `R2` is {file0, file1}, and rule `R3` is {file2}. Because both rule `R2` and `R3` have rule `R4` as one of their subrules, during the preorder graph traversal, the *fileSet* of rule `R4` is updated to the union of their *fileSet*s as {file0, file1, file2}. So, after the first step, every rule's *fileSet* consists of the IDs of all the files containing the string represented by that rule. The second step goes through each rule and builds the inverted indices by outputting the *fileSet* of each word that appears on the right-hand side of that rule.

**Algorithm 3** `invertedindex-V1`

```
1: G = LoadMergeGraph(I)
2: init workQue to empty
3: for each file i in the root rule 0 do
4:     initLevel2Nodes()▷ Insert file info from the root and
   insert level-2 nodes into workQue
5: end for
6: while !workQue.empty() do
7:     head = workQue.deque()
8:     for each subRule i in rule head do
9:         for each file j in rule i do
10:            if j not in rule[i].fileSet then
11:                rule[i].fileSet.insert(j)
12:            end if
13:        end for
14:        rule[i].updates++
15:        if rule[i].updates == rule[i].inEdges then
16:            workQue.enque(i)
17:        end if
18:    end for
19: end while
20: for each rule i in rules do
21:    for each word j in rule[i] do
22:        for each file k in rule[i] do
23:            insert (j, k) into result
24:        end for
25:    end for
26: end for
```

The algorithm of preorder *inverted index* is shown in Algorithm 3. Initialization part transmits the file information to the level-2 nodes, and places the level-2 nodes into the queue (lines 3 to 5). After initialization, level-2 nodes are ready to propagate the file information to the nodes in lower levels. Next, the algorithm propagates the file information so that each rule contains the file IDs it appears in (lines 6 to 19). It uses a queue: each rule transmits its file IDs to its children (lines 9 to 13), and places the children that finish gathering file IDs into the queue (lines 14 to 17). Finally, the algorithm goes through the words and files for each rule to build the word-to-file index (lines 20 to 26).

The complexity of Algorithm 3 is dominated by two parts: 1) transmitting the file information from the root to all the children, and 2) collecting the word-to-file relations from each rule to the global table. For the first part, in the root, each rule element $h_i$ belongs to a file, and the file information needs to be transmitted to all the reachable nodes of rule $h_i$. Assuming the number of rule elements in the root is $h_G$, and the number of the reachable edges of $h_i$ is $u_i$. Then, the time complexity of the first part is $O(\sum_{i=0}^{h_G-1} u_i)$. For the second part,

**Algorithm 4** `invertedindex-V2`

```
1: G = LoadMergeGraph(I)
2: for each subrule i in the root rule 0 do
3:     postorderTraverse(i)
4: end for
5: for each file i in the root rule 0 do
6:     generateIndex(i, result)        ▷ Using root and level-2 nodes to generate the result
7: end for
8: procedure postorderTraverse(id)
9:     for each subRule i in rule id do
10:        if !done[i] then
11:            postorderTraverse(i)
12:        end if
13:    end for
14:    for each subRule i in rule id do
15:        for each word w in the locTbl of rule i do
16:            rule[id].locTbl[w] = true
17:        end for
18:    end for
19:    done[id] = true
20: end procedure
```

assume that node $r_i$ belongs to $v_i$ files, and the number of elements in node $r_i$ is $n_i$. Then, the time complexity of the second part is $O(\sum_{i=0}^{k-1}(v_i * n_i))$. Therefore, the time complexity of Algorithm 3 is $O(\sum_{i=0}^{h_G-1} u_i + \sum_{i=0}^{k-1}(v_i * n_i))$. The space complexity of Algorithm 3 is also $O(s+g+k*l_{2lev})$, where $s$ is the size of the DAG, $g$ is the global table size, $k$ is the number of nodes in the DAG, and $l_{2lev}$ is the average size of the double-layered bitmaps (detailed in Section 4.5).

The *postorder* design recursively folds the set of words covered by a node into the word sets of their parent node. The folding follows a postorder traversal of the graph and stops at the immediate children of the root node (called *level-2 nodes*.) The result is that every level-2 node has a *wordSet* consisting of all the words contained by the string represented by that node. From the root node, it is easy to label every level-2 node with a *fileSet*—that is the set of files that contain the node (and hence each word in its *wordSet*). Going through all the level-2 nodes, the algorithm can then easily output the list of containing files for each word, and hence yield the inverted indices.

Algorithm 4 depicts postorder *inverted index*. It consists of two main steps. The first step (lines 2 to 4) performs word transmission in postorder to the level-2 nodes. Different from Algorithm 2 of `wordcount-V2`, the algorithm only needs to record the word without its frequency. Because level-2 nodes contain all the word information needed, the second step only needs the root rule and the level-2 nodes to generate the inverted index (lines 5 to 7).

The complexity of Algorithm 4 is dominated by two parts: 1) transmitting the word set from the leaf to the level-2 nodes, and 2) collecting the word-to-file relations from the root and the level-2 nodes. For the first part, the process is similar to Algorithm 1 and has the same complexity of $O(n_e * k + \sum_0^{k-1}(n_i * e_i))$, where $n_e$ is the average number of elements for the nodes, $k$ is the number of nodes, $n_i$ is the number of elements in the local word table of node $i$, and $e_i$ is the number of edges in the reverse reachable edge set of node $i$ in the DAG. For the second part, the number of words contained in each level-2 node depends on the input, and in the worst-case scenario, each level-2 node includes all vocabularies, whose number is $y$. Hence, the complexity of the second part is $h_G * y$, where $h_G$ is the number of rule elements in the root. Therefore, the time complexity of Algorithm 4 is $O(n_e * k + \sum_0^{k-1}(n_i * e_i) + h_G * y)$. The space complexity of Algorithm 4 is $O(s+g+k*l)$, where $s$ is the size of the DAG, $g$ is the global table size, $k$ is the number of nodes in the DAG, and $l$ is the average size of the local tables.

The relative performance of the two designs depends on the dataset. For a dataset with many small files, the preorder design tends to run much slower than postorder (e.g., 1.2X versus 1.9X speedup over processing the original dataset directly on dataset D in Section 7.1, *NSF Research Award Abstracts* dataset [51]), because the file sets it propagates are large. On the other hand, for a dataset with few large files, the preorder design tends to be a better choice as the postorder design has to propagate large *wordSets*.

It is worth noting that *word count* can also be implemented in both preorder and postorder, and preorder is a more efficient choice.

*Guidelines and Software Module*

Our experience leads to the following two guidelines for implementing TADOC.

**Guideline I:** Try to minimize the footprint size of the data propagated across the graph during processing.

**Guideline II:** Traversal order is essential for efficiency. It should be selected to suit both the analytics task and the input datasets.

These guidelines serve as principles for developers to follow during their implementations of the solutions for their specific analytics tasks.

Traversal order is worth further discussion. The execution time with either order mainly consists of the computation time $t_{compute}$ and the data propagation time $t_{copy}$. The former is determined by the operations performed on a node, while the latter by the amount of data propagated across nodes. Their values in a given traversal order are affected by both the analytics task and the datasets. Directly modeling $t_{compute}$ and $t_{copy}$ analytically is challenging.

We instead provide support to help users address the challenge through machine learning models. For a given analytics problem, the developer may create multiple versions of the solution (e.g., of different traversal orders). We use a decision tree model to select the most suitable version. To build the model, we specify a list of features that potentially affect program performance. According to the decision tree algorithm, these features are automatically selected and placed on the right place of the decision tree via the training process. For training data, we use some small datasets that have similar characteristics to the target input.

We develop a software module, *OrderSelector*, which helps developers to build the decision tree for version selection. The developer can then specify these versions in the configuration file of *OrderSelector* as candidates, and provide a list of representative inputs on which the program can run. They may also specify some (currently Python) scripts for collecting certain features of a dataset that are potentially relevant to the selection of the versions. This step is optional as *OrderSelector* has a set of predefined data feature collection procedures, including, for instance, the size of an original dataset, the size of its Sequitur CFG, the number of unique words in a dataset, the number of rules, and the number of files. These features are provided as metadata at the beginning of the Sequitur compressed data or its dictionary, taking virtually no time to read. With the configuration specified, *OrderSelector* runs all the versions on each of the representative input to collect their performance data (i.e., running time) and dataset features. It then invokes an off-the-shelf decision tree construction tool (scikit-learn [69]) on the data to construct a decision tree for version selection. The decision tree is then used in the production mode of *OrderSelector* to invoke the appropriate version for a given compressed dataset.

Figure 6 shows the decision tree obtained on *inverted index* based on the measurements of the executions of the different versions of the program on 60 datasets on the single node machine (Table 2). The datasets were formed by sampling the documents contained in the five datasets in Section 7. They have various features: numbers of files range from 1 to 50,000, median file sizes range from 1KB to 554MB, and vocabulary sizes range from 213 to 3.3Million. The decision tree favors postorder traversal when the average file size is small ($<$2860 words) and preorder otherwise. (The two versions of preorder will be discussed in Section 4.5). In five-fold cross validation, the decision tree predicts the suitable traversal order with a 90% accuracy.

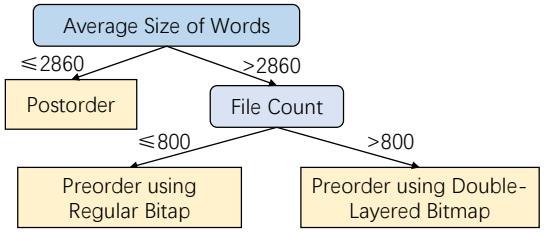


Fig. 6: Decision tree for choosing traversal order.

### 4.3 Coarse-Grained Parallelism and Data Partitioning

To obtain scalable performance, it is important for TADOC to effectively leverage parallel and distributed computing resources. As Section 3 mentions, some dependencies are introduced between processing steps in either preorder or postorder traversals of CFGs, which cause extra challenges for a parallel implementation.

We have explored two ways to handle such dependencies to parallelize the processing. The first is *fine-grained partitioning*, which distributes the nodes of the DAG to different threads, and inserts fine-grained communication and synchronization among the threads to exchange necessary data and results. This method can potentially leverage existing work on parallel and distributed graph processing [35, 74, 18, 57, 55, 91]. For instance, PowerGraph [35], exploits the power-law property of graphs for distributed graph placement and representation, and HDRF [74] is a streaming vertex-cut graph partitioning algorithm that considers vertex degree in placement.

The second is a *coarse-grained partitioning* method. At compression time, this method partitions the original input into a number of segments, then performs compression and analytics on each segment in parallel, and finally assembles the results if necessary.

The coarse-grained method may miss some compression opportunities that exist across segments (e.g., one substring appears in two segments). However, its coarse-grained partitioning helps avoid frequent synchronization among threads. Our experimental results show that on datasets of non-trivial sizes, the coarse-grained method significantly outperforms the fine-grained method in both performance and scalability. It is also easier to implement, for both parallel and distributed environments. For a parallel system, the segments are distributed to all cores evenly. For a distributed system, they are distributed to all nodes evenly, and then distributed to the cores within each node.

Load balance among threads or processes is essential for high parallel or distributed performance. Thus, the coarse-grained method requires balanced partitioning of input datasets among threads or processes. The partitioning can be done at the file level, but it sometimes requires even *finer* granularity such that a file is

split into several sections, where each section is assigned to a thread or process.

*Guideline and Software Module*

Our experience leads to the following guideline.

**Guideline III:** Coarse-grained distributed implementation is preferred, especially when the input dataset exceeds the memory capacity of one machine; data partitioning for load balance should be considered, but with caution if it requires the split of a file, especially for unit-sensitive or order-sensitive tasks.

Dataset partitioning is important for balancing the load of the worker threads in coarse-grained parallelization. Our partitioning mechanism tries to create subsets of files rather than splitting a file because there is extra cost for handling a split file, especially for unit-sensitive or order-sensitive tasks. To assist with this process, we develop a software module. When the module is invoked with the input dataset (a collection of files) and the intended number of worker threads, it returns a set of partitions and a metadata structure. Resilient distributed dataset (RDD) is the basic fault-tolerant data unit in Spark [93]. The metadata structure records the mapping relations among RDDs, files, and file sections. In the workload partitioning process, file splitting is considered only when a file exceeds a size threshold, $h_{split}$, and causes significant load imbalance (making one partition exceed the average workload per worker by 1/4). $h_{split}$ is defined as $S_{total}/2n_w$, where $S_{total}$ is the total dataset size, and $n_w$ is the number of workers. The module 1) ensures that all workers process similar amounts of work and 2) avoids generating small fragments of a file by tolerating some disparity in the partition sizes. For applications that require additional file or word sequence information, our partitioning mechanism records some extra information, such as which file a section belongs to, the sequence number of the section in the file, and so on. Such information is necessary for a thread to know which section of which file it is processing, which is useful for a later stage that merges the results.

### 4.4 Handling Order Sensitivity

As Section 3 mentions, tasks that are sensitive to the appearance order of words pose some special challenges. *Sequence count*, for instance, requires extra processing to handle 1) sequences that may span across multiple Sequitur rules (i.e., nodes in the DAG) and 2) order of words covered by different rules. The order sensitivity challenge (detailed in Section 3.2) 1) calls for certain constraints on the visiting order of the rules in the Sequitur grammar, and 2) demands the use of extra data structures to handle sequences across rules.

In our explorations, we found that the order sensitivity challenge can be addressed through a two-level table design with a depth-first graph traversal. The depth-first traversal of the graph ensures that the processing of the data observes the appearing order of words in the original dataset. During the traversal, we use a global table to store the results that require cross-node examinations, and a local table to record the results directly attainable from the right hand side of the rule in a node. Such a design allows the visibility of results across nodes, and at the same time, permits reuse of local results if a rule appears many times in the dataset.

We take *sequence count* as an example to illustrate our solution. Algorithm 5 shows the pseudo-code. The Sequitur design decides that the scanning process is similar to the depth-first graph traversal, but the difference is that a sequence counting is integrated into the scanning process. The general idea is to perform a depth-first traversal (line 5); the word sequences that cross rules are stored in the global table, while the word sequences within a rule are stored in the local table of each rule. When the traversal is finished, we integrate the local tables from different rules to the global table (lines 6-7). The `seqCount` function is used to process the rules, while the `process` function is used to process the words. The depth-first graph traversal is embodied by the recursive function `seqCount` (lines 10 and 15 in Algorithm 5). It uses an $l$-element first-in first-out queue (`q`) to store the most-recently-seen $l$ words. In function `process`, the most recent word is pushed into `q`, and this newly formed sequence in `q` is then processed, resulting in an increment in the counters in either the local table `locTbl` (line 27) if the sequence does not span across rules, or otherwise, in the global table `gloTbl` (line 24). The traversal may encounter a rule multiple times if the rule or its ancestors are referenced multiple times in the DAG. The Boolean variable `locTblReady` of a rule tracks whether the `locTbl` of the rule is ready to be reused, thus saving time. Note that in line 21, we also need to record the rule $r$, since we need to identify whether the words in $q$ are from multiple rules.

Figure 7 demonstrates how Algorithm 5 works on an input word sequence whose DAG is shown in Figure 5. The words in the first 3-word sequence ① correspond to two different rules in the DAG (R1 and R4). This sequence is a cross-node sequence and the algorithm stores its count into a global table. The next 3-word sequence ② corresponds to only R4, and is hence counted in a local table. The next two sequences ③, ④ both correspond to two instances of R4, and are both cross-node sequences. Thus, they are counted in the global table. The bottom sequence ⑤ is the same as the second sequence ②; the algorithm does *not* recount this

**Algorithm 5** Count $l$-word Sequences in Each File

```
1: G = LoadData(I)   ▷ load compressed data I; each rule has an empty locTbl and a false boolean locTblReady
2: allocate gloTbl and an l-element long FIFO queue q
3: for each file f do
4:     s = segment(f, G.root)   ▷ Get a segment of the right-hand side of the root rule covering file f (e.g., first three nodes in Figure 5 for file0)
5:     seqCount(s)
6:     calfq(s) ▷ calculate the frequency fq of each rule in s
7:     cmb(s)▷ integrate into gloTbl the locTbl (times fq) of each rule subsumed by s
8:     output gloTbl and reset it and q
9: end for
10: function seqCount(s)
11:     for each element e in s from left to right do
12:         if e is a word then
13:             process(e, s)
14:         else
15:             seqCount(e) ▷ recursive call that materializes depth-first traversal of G
16:         end if
17:     end for
18:     s.locTblReady = true
19: end function
20: function process(e, r)
21:     q.enqueue(e, r)   ▷ evict the oldest if full
22:     return if q is not full   ▷ Need more words to form an l-element sequence
23:     if words in q are from multiple rules then
24:         gloTbl[q.content] + +
25:     else
26:         if !r.locTblReady then   ▷ avoid repeated processing
27:             r.locTbl[q.content] + +
28:         end if
29:     end if
30: end function
```

sequence, but directly reuses the entry in the local table of R4.

The key for the correctness of Algorithm 5 is that the depth-first traversal visits *all* words in the *correct* order, i.e., the original appearance order of the words. We prove it as follows ("content of a node" means the text that a node covers in the original document.)

*Lemma 1*: If the content of every child node of a rule $r$ is visited in the correct order, the content of $r$ is visited in the correct order.

*Proof:* Line 11 in Algorithm 5 ensures that the elements in rule $r$ are visited in the correct order. The condition of this lemma ensures that the content inside every element (if it is a rule) is also visited in the correct order. The correctness of the lemma hence follows.

*Lemma 2*: The content of every leaf node is visited in the correct order.

*Proof:* A leaf node contains no rules, only words. Line 11 in Algorithm 5 ensures the correct visiting order of the words it contains. The correctness hence follows.

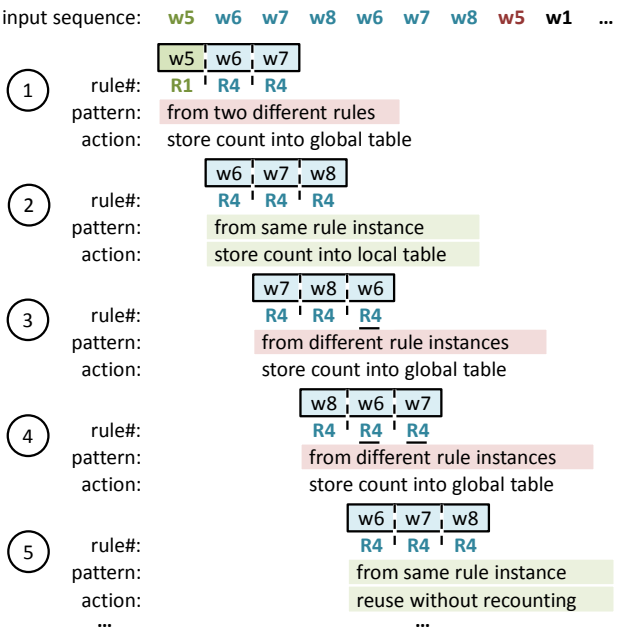


Fig. 7: Illustration of how Algorithm 5 processes an input sequence (DAG in Figure 5) for counting 3-word long sequences.

*Lemma 3*: The depth-first traversal by Algorithm 5 of a DAG G visits all words in an order consistent with the appearance order of the words in the original document G.

*Proof:* A basic property of a DAG is that it must have at least one topological ordering. In such an ordering, the starting node of every edge occurs earlier in the ordering than the ending node of the edge. Therefore, for an arbitrary node in G, its children nodes must appear *after* that node in the topological ordering of G. Let $n_{-1}$ be the last non-leaf node in the ordering. *Lemma 2* entails that all the nodes after $n_{-1}$ must be visited in the correct order as they are all leaf nodes, and *Lemma 1* entails that the content of $n_{-1}$ must be visited in the correct order. The same logic leads to the same conclusion on the second to the last non-leaf node, then the third to the last, and so on, until the first node—that is, the root node. As the content of the root node is the entire document, *Lemma 3* follows, by induction.

With *Lemma 3*, it is easy to see that all $l$-long sequences in the original document goes through the FIFO queue `q` in Algorithm 5. The algorithm uses the two tables `locTbl` and `gloTbl` to record the counts of every sequence in `q`. Function `cmb` folds all information together into the final counts.

The computational complexity of Algorithm 5 depends mainly on two functions, `seqCount` and `process`. The complexity of `seqCount` is determined by the total number of times rules are visited, which is also the total number of times edges are traversed in the DAG.

In reality, especially with coarsening to be described in Section 4.5, the overhead of `seqCount` is much smaller than the overhead of `process`. The complexity of Algorithm 5 is practically dominated by `process`, which has a complexity of $O(w)$. $w$ is the number of words in the input documents. The time savings of Algorithm 5 over the baseline of direct processing on the original data (i.e., without using our method) comes from avoiding repeatedly counting the sequences that do *not* span across rules. Thus, the amount of savings is proportional to $m/n$, where $m$ is the number of repeated local sequences and $n$ is the total number of sequences. The space complexity of Algorithm 5 is $O(s + g + k * l)$, where, $s$ is the size of the DAG, $g$ is the global table size, $k$ is the number of nodes in the DAG, and $l$ is the average size of the local tables.

*Guideline*

**Guideline IV:** For order-sensitive tasks, consider the use of depth-first traversal and a two-level table design. The former helps the system conform to the word appearance order, while the latter helps with result reuse.

The global and local tables can be easily implemented through existing template-based data structures in C++ or other languages. Hence, there is no specific software module for the application of this guideline. The coarsening module we describe next provides important performance benefits on top of this guideline.

### 4.5 Other Implementation-Level Optimizations

We introduce three extra optimizations. They are mostly implementation-level features that are useful for deploying TADOC efficiently.

**Double-layered bitmap**. As Guideline I says, minimizing the footprint of propagated data is essential. In our study, we find that when dealing with unit-sensitive analytics (e.g., *inverted index* and *sequence count*), double-layered bitmap is often a helpful data structure for reducing the footprint.

Double-layered bitmap in this study is a data structure that includes level one and level two, and is used to store information about which files the rule belongs to in an efficient manner. Level one is a data structure used to point to the locations for storing the file information in level-2 bitmaps for a given rule. Level two is the data structure of bit vectors that actually store which files the rule belongs to. In detail, as Figure 8 illustrates, level two contains a number of $N$-bit vectors (where $N$ is a configurable parameter), while level one contains a pointer array and a level-1 bit vector. The pointer array stores the starting address of the level-2 bit vectors, while the level-1 bit vector is used for fast checks to determine which bit vector in level 2 contains the bit corresponding to the file that is being queried. If a rule is contained in only a subset of files whose bits correspond to some bits in several level-2 vectors, the level two of the bitmap associated with that rule would then contain only those several vectors, and most elements in the pointer array would be null. The number of elements in the first-level arrays and vectors of the double-layered map is only $1/N$ of the number of files. For example, assume that $N$ is four and there are 12 files. The rule belongs to $file0$, $file1$, $file3$, $file4$, and $file5$. Then, in level one, $bit0$ and $bit1$ are *true*, while $bit2$ is *false*; $P0$ and $P1$ have pointer addresses and the value of $P2$ is $NULL$. In level two, the first bit vector is "1101" ($file0$, $file1$, and $file3$), the second bit vector is "1100" ($file4$ and $file5$), and there is no bit vector that relates to $P2$.

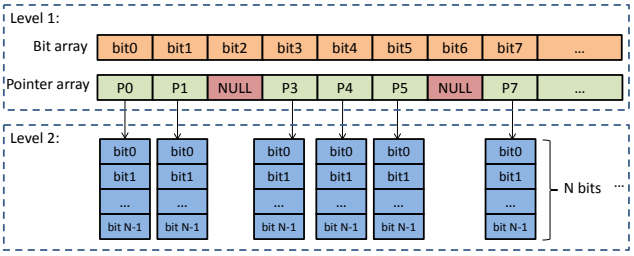


Fig. 8: A double-layered bitmap for footprint minimization and access efficiency.

Time-wise, the double-layered bitmap provides most of the efficiency benefits of the one-level bit vector compared to the use of the set mechanism. Even though it incurs one possible extra pointer access and one extra bit operation compared to the use of one-level bit vectors, its total memory footprint is much smaller, which contributes to better cache and TLB performance. As Figure 6 shows, the selection between the use of single-layer bitmap and double-layered bitmap can be part of the traversal order selection process. The decision tree in Figure 6 favors double-layered bitmap when the average file size is greater than 2860 words and the number of files is greater than 800. (The benefits are confirmed experimentally in Section 7).

It is easy to see that the double-layered bitmap can be used in other applications with unit sensitivity as well. If the unit is a class of articles, for instance, one just needs to change the semantics of a bit to represent the class.

**Double compression**. *Double compression* is an optimization we have found helpful for the compression step. Compared with some other compression algorithms, our Sequitur-based method is based on words and it does *not* always obtain the highest compression rates. To solve this issue, we first compress the original dataset with Sequitur and then run "gzip" on the output of

Sequitur. Note that "gzip" can be replaced with other methods with high compression rates, such as `zstd` [7] (`zstd` has a good compression ratio and is extremely fast at decompression). Considering the current universality that almost all platforms are equipped with `gzip`, we shall still keep `gzip` as our default double-compression tool, but other compression tools are also supported in this work. The result is often even more compact than the "gzip" result on the original dataset. To process the data, one only needs to decompress the "gzip" result to recover the Sequitur result. Because Sequitur result is usually much smaller than the original dataset, it takes much less time to recover the Sequitur result than the original dataset does. The decompression of the "gzip" result adds only a very small marginal overhead, as shown later.

**Coarsening**. The third optimization relates to data loading time. It is called *coarsening*, a transformation to the Sequitur DAG. Through it, the nodes or edges in the graph can represent the accumulated information of a set of nodes or edges. Specifically, we have explored two coarsening methods: edge merging and node coarsening. *Edge merging* merges the edges between two nodes in the DAG into one, and uses a weight of the edge to indicate the number of original edges. Merging loses the order of words, but helps reduce the size of the graph and hence the number of memory accesses in the graph traversal. It is helpful to analytics tasks that are insensitive to word order (e.g., *word count* and *inverted index*). *Node coarsening* inlines the content of some small rules (which represent short strings) into their parent rules; those small nodes can then be removed from the graph. It reduces the size of the graph, and at the same time, reduces the number of substrings spanning across nodes, which is a benefit especially important for analytics on word sequences (e.g., *sequence count*). Coarsening adds some extra operations, but the time overhead is negligible if it is performed during the loading process of the DAG. On the other hand, it can save memory usage and graph traversal time, as reported in the next section.

*Guideline and Software Module*

**Guideline V:** When dealing with analytics problems with unit sensitivity, consider the use of double-layered bitmap if unit information needs to be passed across the CFG.

To simplify developers' job, we create a collection of double-layered bitmap implementations in several commonly used languages (Java, C++, C). Developers can reuse them by simply including the corresponding header files in their applications.

Besides double-layered bitmap, another operation essential for handling unit sensitivity is the insertion of special markers into the documents to indicate unit boundaries when we do the compression as Section 4.2 mentions.

**Guideline VI:** *Double compression* and *coarsening* help reduce space and time cost, especially when the dataset consists of many files. They also enable that the thresholds be determined empirically (e.g., through decision trees).

We create two software modules to assist developers in using our guideline. One module is a library function that takes original dataset as input, and conducts Sequitur compression on it, during which, it applies dictionary encoding and *double compression* automatically. In our implementation, this module and the partitioning module mentioned in Section 4.3 are combined into one compression module such that the input dataset first gets integer indexed, then partitioned, and finally compressed. The combination ensures that a single indexing dictionary is produced for the entire dataset; the common dictionary for all partitions simplifies the result merging process.

Our other module is a data loading module. When this module is invoked with coarsening parameters (e.g., the minimum length of a string a node can hold), it loads the input CFG with coarsening automatically applied.

### 4.6 Short Summary

The six guidelines described in this section capture the most important insights we have learned for unleashing the power of TADOC. They provide the solutions to all the major challenges listed in Section 3.2: Marker insertion described in Section 4.2 and Guideline V together address unit sensitivity, Guideline IV order sensitivity, Guideline II data attributes challenge, Guideline III parallelism barriers, while Guidelines I and VI provide general insights and common techniques on maximizing the efficiency. The described software modules are developed to simplify the applications of the guidelines. They form part of the `CompressDirect` library, described next.

## 5 CompressDirect Library

We create a library named `CompressDirect` for two purposes. The first is to ease programmers' burden in applying the six guidelines when developing TADOC for an analytics problem. To this end, the first part of `CompressDirect` is the collection of the software modules described in the previous section. The second purpose is to provide a collection of high performance implementations of some frequently-performed document analytics tasks, which can directly help many existing applications.

Specifically, the second part of `CompressDirect` consists of six high-performance modules. `Word count` [9] counts the number of each word in all of the input documents. `Sort` [40] sorts all the words in the input documents in lexicographic order. `Inverted index` [9] generates a word-to-document index that provides the list of files containing each word. `Term vector` [9] finds the most frequent words in a set of documents. `Sequence count` [9] calculates the frequencies of each three-word sequence in every input file. `Ranked inverted index` [9] produces a list of word sequences in decreasing order of their occurrences in each document. These modules are essential for many text analytics applications.

For each of these modules, we implement three versions: sequential, parallel, and distributed. The first two versions are written in C/C++ (with Pthreads [67] for parallelization), and the third is in Scalar on Spark [93]. Our implementation leverages the functions contained in the first part of the library, which are the software modules described in Section 4. A DAG is loaded into memory before it is processed. Large datasets are partitioned first with each partition generating a DAG that fits into the memory. The data structures used for processing are all in memory. Using a Domain Specific Language may further ease the programming difficulty, as elaborated in a separate work [98]. We next report the performance of our implementations.

## 6 Supporting Advanced Document Analytics

In this section, we explore opportunities to apply TADOC to advanced document analytics. We apply TADOC to three advanced document analytics applications: *word co-occurrence* [58,72], *term frequency-inverse document frequency* (TFIDF) [41], *word2vec* [79,3], and *latent Dirichlet allocation* (LDA) [12].

### 6.1 Word Co-Occurrence

Word co-occurrence is the frequency of the occurrence of two words alongside each other in a text corpus. Because it can be regarded as a semantic proximity indicator, word co-occurrence is widely used in linguistics, content analysis, text mining, and thesauri construction. In general, word co-occurrence research aims to analyze similarities between word pairs and patterns, and thus discovers latent linguistic patterns and structures in representations. Word co-occurrence can be transformed into a word co-occurrence matrix, as shown in Figure 9. The rows and columns represent unique words, and the numbers in the matrix denote the frequency at which $word_i$ co-occurs with $word_j$. For example, in Figure 9, $word_0$ co-occurs with $word_1$ six times but does not co-occur with $word_2$.

The TADOC technique can be applied as a data provider. Because word co-occurrence only pertains to

| | word0 | word1 | word2 | word3 | ··· | word n |
|---|---|---|---|---|---|---|
| word0 | | 6 | 0 | 0 | ··· | 8 |
| word1 | 6 | | 1 | 11 | ··· | 0 |
| word2 | 0 | 1 | | 0 | ··· | 0 |
| word3 | 0 | 11 | 0 | | ··· | 9 |
| ··· | ··· | ··· | ··· | ··· | ··· | ··· |
| word n | 8 | 0 | 0 | 9 | ··· | |

Fig. 9: Word co-occurrence matrix example.

the frequency of adjacent words, we can reuse the word co-occurrences in each rule. A similar process also occurs in *sequence count*. Several optimizations can occur in this process. In linguistics, due to grammar and rules, many words do not appear together; hence, most of the elements in a matrix could be zero, and the word co-occurrence data can be stored in a sparse matrix format, such as the compressed sparse row (CSR) and coordinate (COO) storage format [96]. In addition, a word co-occurrence matrix is a symmetric matrix when the word order is unnecessary, in which case we store only the upper triangular part of the matrix. We only need to coordinate our program interface with the corresponding word co-occurrence program interface.

In this paper, we use the word co-occurrence in GloVe [72] for validation. The word co-occurrence matrix generated by this implementation is a mixture of dense and sparse matrices, as shown in Figure 10. The dense part represents frequent words that have a large number of occurrences; these words are more likely to co-occur with other words, and are therefore stored in a dense matrix. The parameter *vocab_size* denotes the size of the dense matrix, which can be adjusted by users. The sparse part represents words that have a small number of occurrences, and therefore uses a sparse matrix format. Notably, GloVe uses a coordinate (COO) format for these sparse matrices.

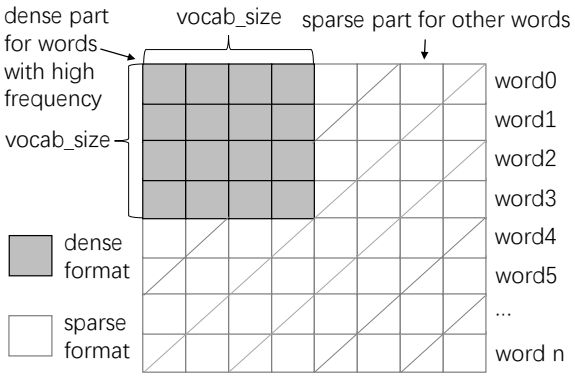


Fig. 10: Word co-occurrence matrix in GloVe.

TADOC can be applied in the construction of dense and sparse matrices. To separate words and build the two matrix parts, the co-occurrence implementation first

counts the frequency of each word, and then calculates which words should be arranged in the dense matrix part. The `CompressDirect` library directly provides word counts from the compressed format, which is more efficient than providing these counts from the original uncompressed data. Because the word count in `CompressDirect` reuses the redundant information, it saves both computation and IO time.

### 6.2 TFIDF

TFIDF [41] is a statistical method used to evaluate the importance of a word to a document in the corpus. The importance of a word increases proportionally with the frequency of the word in the document, but it decreases inversely with the frequency of the word in the corpus. TFIDF is a commonly used weighting technique for information retrieval and text mining, and has been widely used in search engines as a measurement of the correlation between text and users. Figure 11 shows an example of TFIDF; $score_{i,j}$ exhibits the importance of $word_i$ to $document_j$.

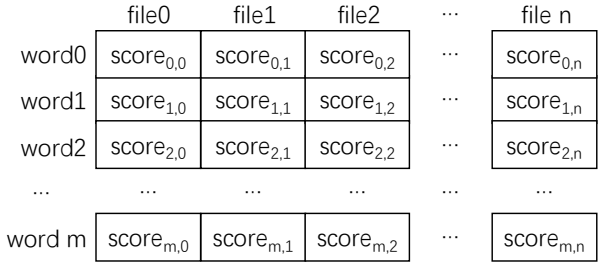


Fig. 11: TFIDF model.

TFIDF consists of two parts: term frequency (TF), and inverse document frequency (IDF). $TF(i,j)$ refers to the frequency of a given word $i$ that appears in document $j$. $IDF_i$ is a measurement of the general importance of word $i$, as shown in Equation 1. In Equation 1, $|D|$ is the number of documents, and $DF_i$ is the number of documents in which word $i$ appears at least once. Given this definition of IDF, when a word occurs in a small number of files, its importance is large; when the word occurs in many files, its importance is small.

$$IDF_i = log(\frac{|D|}{DF_i}) \tag{1}$$

In the TFIDF algorithm, the $score_{i,j}$ of word $i$ to document $j$ is calculated using Equation 2. Intuitively, word $i$ is important for document $j$ if the former occurs frequently in the latter, but the importance of word $i$ decreases if the word occurs in many documents.

$$score_{i,j} = TF(i,j) \cdot IDF_i \tag{2}$$

TADOC can be used in the TFIDF algorithm. The library `CompressDirect` generates the TF for each document, and the TF generation process is similar to that of the `term vector`. Because `CompressDirect` supports the `inverted index`, we can first execute the `inverted index` to calculate $DF$, and then use the intermediate results to calculate the $IDF$ for each file.

To validate the efficiency of TADOC on TFIDF, we implement TFIDF [41] using both the original data (baseline) and the compressed data. The TFIDF algorithm can be divided into two stages. In the first stage, TF and IDF are calculated. Specifically, we calculate the word frequency in each document in the dataset to obtain TF; meanwhile, we record the inverted word-to-document index, which can be used to obtain IDF. In the second stage, the TFIDF values are calculated using both TF and IDF, which are represented as the scores in Figure 11.

The difference between the use of TFIDF with and without TADOC lies in the first stage. In the baseline, we need to process words sequentially in the original files to generate TF and IDF. In the `CompressDirect` version, the word frequencies and inverted index of each file are obtained by traversing the DAG, during which deduplication occurs in the reuse of rules, thus saving both space and computation time. The second stages of the baseline and the `CompressDirect` version are the same.

### 6.3 Word2vec

Word2vec [79] is a shallow two-layer neural network that converts words from documents to feature vectors, as shown in Figure 12. The input of word2vec is a bag of text documents, and its outputs are feature vectors used to describe the words in the text corpus. The vector output of word2vec can be used as input in many applications, such as long short-term memory (LSTM) [22] and recommendation systems [88], since this output represents words in a limited number of feature dimensions by numbers. Originally, word2vec was designed to process text data, but its applicability has been extended beyond the text corpus scope; word2vec has applications in non-text patterns such as genes [43], code [78], and social media [92]. In this work, we mainly concentrate on the text document corpus.

The goal of word2vec is to use a vector to represent each word; the dimension of the vector is limited, but the vector can represent the meaning of the word with precise properties. As shown in Figure 12, the input to the network is a one-hot encoded vector for a word; the length of the vector is equal to the number of unique words, and the vector of the index represents the corresponding word in the vocabulary. The neurons in the hidden layer represent different features, so the neuron size is equal to the size of the word vector. The output layer can be regarded as a probability vector; each el-

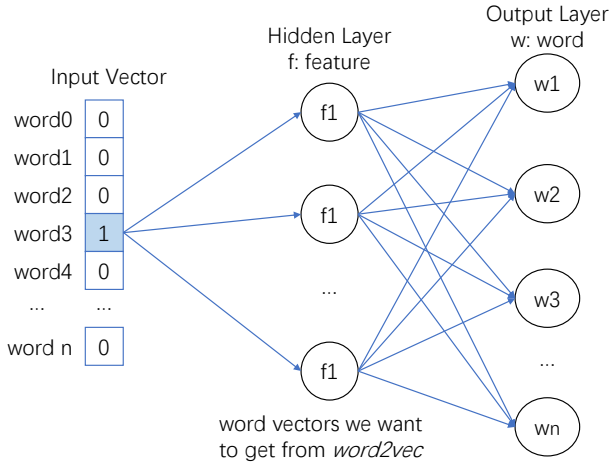


Fig. 12: Word2vec example.

ement represents the probability that a word appears around the input word; hence, the probability vector length is equal to the vocabulary size. Notably, once the training has finished, we only need the word vector trained in the hidden layer, and the neuron network itself is useless; further studies can be conducted with the generated word vectors.

TADOC can be applied in the construction of the input vector. Building the input vector from the compressed data with `CompressDirect` is more efficient than building the input vector from the original data, not only due to the smaller storage size. In real word2vec implementations, the input vector is more complex than the input vector illustration in Figure 12, and words need to be encoded according to different specifications. Those coding specifications are usually related to word frequency, and TADOC can efficiently provide the required word frequencies.

We use the word2vec in [3] for validation. In this implementation, before training, word2vec has a preprocessing step, during which word2vec loads input data into the system and then encodes words. In the encoding process, Huffman coding [86] is used for the words; first, words are sorted according to their frequency, and then shorter codes are assigned to words with higher frequencies. In the next training step, Huffman codes are used instead of the words. As for TADOC, Sequitur-compressed data can be used as input in the preprocessing step of word2vec. Because word2vec needs the word frequency for Huffman coding, the `CompressDirect` library provides the word counts in addition to the word sequence.

### 6.4 Latent Dirichlet Allocation (LDA)

LDA [12] is a generative probabilistic model that is popular in machine learning and natural language processing, where it is used to provide the subjects of documents in a probability form. LDA is an unsupervised learning algorithm; it does not require manual labeling. The only information that users need to provide is the number of specified topics for the given documents. In addition, for each topic, LDA can provide relevant words (these words define the abstracted topic). The graphical model representation of LDA is shown in Figure 13.

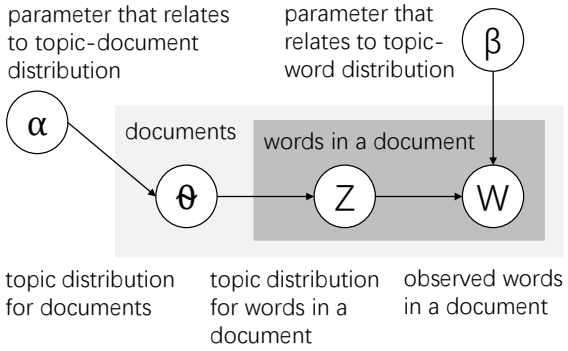


Fig. 13: LDA model. The parameters $\alpha$ and $\beta$ usually have default values, so only the documents and the number of specified topics are input to LDA.

LDA assumes that documents are generated by latent topics, and each topic is characterized by a word distribution, which explains why a bag of words appears in a given document. The LDA model consists of five major components. The parameter $\alpha$ relates to the topic-document distribution; the topic distribution for documents, $\theta$, follows a Dirichlet distribution with parameter $\alpha$. The topic distribution for words, $Z$, follows a multinomial distribution of $\theta$. The words observed in a document, $W$, are determined by both $Z$ and $\beta$. In general, $\alpha$ and $\beta$ are corpus-level parameters, $\theta$ is a document-level variable, and $Z$ and $W$ are word-level variables that can be sampled in each document. In the training process of LDA, the posterior distribution of latent variables of $\theta$ and $Z$ are estimated given $W$, $\alpha$, and $\beta$.

The compressed data using TADOC can be directly applied to the input preprocessing of LDA. The inputs to LDA are isolated words. LDA is based on the "bag-of-words" assumption, that is, the order of words in a document can be neglected. TADOC generates word counts by efficiently traversing the DAG, which means that the required input data is provided in a very effective manner.

We use the LDA in [53] as the evaluation platform. In this LDA implementation, the training process consists of two stages. The first stage is a preprocessing stage; the original input needs to be preprocessed into a sparse representation. The sparse representation stores each document in one line. In each line, each word and its word count are stored sequentially as $<word_i,count_i>$.

The second stage is a training process. This stage uses the preprocessed data in the first stage and trains the probabilistic model mentioned in Section 6.4 to estimate parameters. In addition, the LDA implementation [53] provides a parameter that controls the number of iterations in the training process; in our evaluation, we set this parameter to ten.

TADOC can be applied in the first stage. The baseline version processes the original input data into the required sparse format, while the `CompressDirect` version provides the word counts in the required format directly from the compressed data. In addition, because LDA adopts a "bag-of-words" assumption in which word order does not need to be maintained, we can simply use the word counts in `CompressDirect` in this stage. For the second stage, the baseline and the `CompressDirect` version have the same procedure.

### 6.5 Other Advanced Document Analytics

TADOC can be applied to other advanced document analytics. To realize the high-level goals of advanced document analytics such as LSTM, the original raw text first needs to be converted into a vector format. For this purpose, word2vec can be used with TADOC, as we already demonstrated. After the conversion, these vectors have high-level usage models, which are independent from our technique. These independent techniques only use in-memory data structures without accessing the input raw data again, which is orthogonal to the problem we are addressing. Therefore, our technique can be used to support these advanced uses of document analytics, especially at the raw text processing stage.

## 7 Evaluation

Using the six algorithms listed at the end of the previous section, we evaluate the efficacy of the proposed Sequitur-based document analytics for both space and time savings. The baseline implementations of the six algorithms come from existing public benchmark suites, `Sort` from HiBench [40] and the rest from Puma [9]. We report performance in both sequential and distributed environments. For a fair comparison, the original and optimized versions are all ported to C++ for the sequential experiments and to Spark for the distributed experiments.

The benefits are significant in both space savings and time savings. Compared to the default direct data processing on *uncompressed* data, our method speeds up the data processing by more than a factor of two in most cases, and at the same time, saves the storage and memory space by a factor of 6 to 13. After first explaining the methodology of our experimental evaluation, we next report the overall time and space savings, and then describe the benefits coming from each of the major guidelines we have described earlier in the paper.

### 7.1 Methodology

**Evaluated Methods** We evaluate three methods for each workload-dataset combination. The "baseline" method processes the dataset directly, as explained at the beginning of this section. The "CD" method is our version using *TADOC*. The input to "CD" is the dataset compressed using "double compression" (i.e., first compressed by Sequitur then compressed by Gzip). The "CD" method first recovers the Sequitur compression result by undoing the Gzip compression, and then processes the Sequitur-compressed data directly. The measured "CD" time covers all the operations. The "gzip" method represents existing decompression-based methods. It uses Gzip to compress the data. At processing time, it recovers the original data and processes it.

**Datasets** We use five datasets for evaluations, shown in Table 1. They consist of a range of real-world documents of varying lengths, structures and content. The first three, `A, B, C`, are large datasets from Wikipedia [5], used for tests on clusters. `Dataset D` is NSF Research Award Abstracts (NSFRAA) from UCI Machine Learning Repository [51], consisting of a large number (134,631) of small files. `Dataset E` is a collection of web documents downloaded from the Wikipedia database [5], consisting of four large files.

Table 1: Datasets ("size": original uncompressed size).

| 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 |

The sizes shown in Table 1 are the original dataset sizes. They become about half as large after dictionary encoding (Section 2.1). The data *after* encoding is used for all experiments, including the baselines.

**Platforms** The configurations of our experimental platforms are listed in Table 2. For the distributed experiments, we use the `Spark Cluster`, a 10-node cluster on Amazon EC2 [1], and the three large datasets. The cluster is built with an HDFS storage system [15]. The Spark version is 2.0.0 while the Hadoop version is 2.7.0. For the sequential experiments, we use the `Single Node` machine on the two smallest datasets.

### 7.2 Time Savings

#### *7.2.1 Overall Speedups*

Figure 14 reports the speedups that the different methods obtain compared to the default method on the three large datasets `A, B, C`, all run on the Spark Cluster.

Table 2: Experimental platform configurations.

| Platform | Spark Cluster | Single Node |
|---|---|---|
| OS | Ubuntu 16.04.1 | Ubuntu 14.04.2 |
| GCC | 5.4.0 | 4.8.2 |
| Node# | 10 | 1 |
| CPU | Intel E5-2676v3 | Intel i7-4790 |
| Cores/Machine | 2 | 4 |
| Frequency | 2.40GHz | 3.60GHz |
| MemorySize/Machine | 8GB | 16GB |

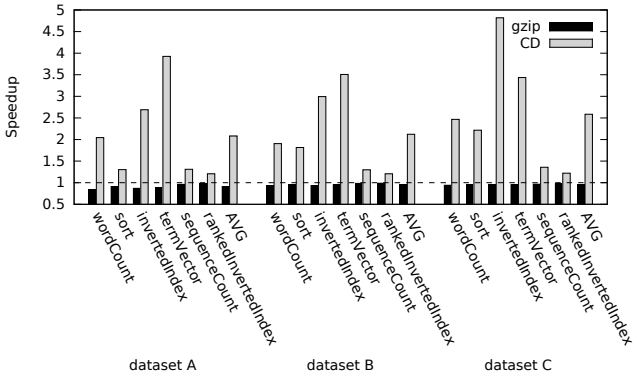


Fig. 14: Performance of different methods on large datasets running on the Spark Cluster, normalized to the performance of the baseline method.

The size of a file in these datasets, in most cases, ranges from 200MB to 1GB. In the implementations of all methods, each file's data form a processing unit (an RDD in Spark), resulting in coarse-grained parallelism. In both the baseline and `CD` methods, each machine in the cluster automatically grabs the to-be-processed RDDs one after another, processes them, and finally merges the results. The two versions differ in whether an RDD is formed on the uncompressed or compressed data, and how an RDD is processed. Because the total size of the uncompressed datasets `B` and `C` exceeds the aggregate memory of the cluster, a newly-loaded RDD reuses the memory of an already-processed RDD.

`Word count` and `sort` use the preorder traversal, `inverted index` and `term vector` use the postorder traversal, and `sequence count` and `ranked inverted index` use the depth-first traversal and the two-level table design of Guideline IV in Section 4.4. Because the three datasets all consists of very large files, the data-sensitivity of order selection does not affect our methods of processing.[1] All the programs use the coarse-grained parallelization. For the coarsening optimization, `word count`, `sort`, `inverted index`, and `term vector` use *edge merging*, because they do not need to keep the order of words. `Sequence count` and `ranked inverted index` use *node coarsening*, because node coarsening reduces the number of substrings spanning across nodes, thereby increasing the reuse of local data. We empirically set 100 as the node coarsening threshold such that each node contains at least 100 items (subrules and words) after coarsening.

The average speedups with our `CD` method are 2.08X, 2.12X, and 2.59X on the three datasets. `Inverted index` and `term vector` show the largest speedups. These two programs are both unit sensitive, producing analytics results for each file. `CD` creates an RDD partition (the main data structure used in Spark) for the compressed results of each file, but the baseline method cannot because some of the original files exceed the size limit of an RDD partition in Spark—further partitioning of the files into segments and merging of the results incur some large overhead. Programs `word count` and `sort` are neither unit sensitive nor order sensitive. `Sort` has some extra code irrelevant to the `CD` optimizations, and hence shows a smaller overall speedup. Programs `sequence count` and `ranked inverted index` are both about word sequences in each file; the amount of redundant computations to save is the smallest among all the programs.

The `gzip` method incurs only 1-14% *slowdown* due to the extra decompression time, and TADOC with double compression can still achieve significant performance benefits compared to the original version without compression, which proves the effectiveness of our double compression techniques.

Figure 15 reports the overall speedups on the two smaller datasets on the single-node server. `CD` provides significant speedups on them as well, while the `gzip` method causes even more slowdown. The reason is that the computation time on the small datasets is little and hence the decompression overhead has a more dominant effect on the overall time. We discuss the time breakdowns in more detail next.

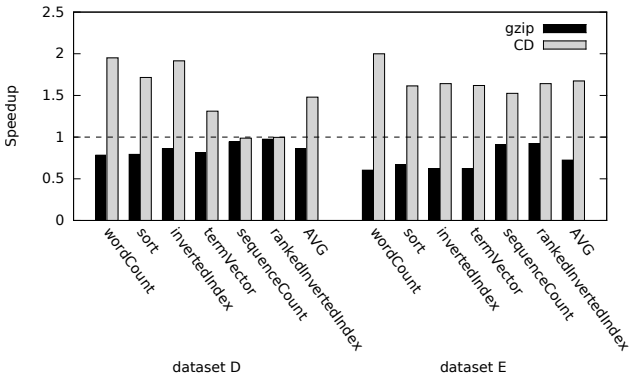


Fig. 15: Performance of different methods normalized to the baseline method on the Single Node machine.

`CD` on the single node server can also benefit from parallelism. With coarse-grained parallelism, the computation time can be further reduced. We implement

[1] Section 7.6 shows the sensitivity on the other two datasets, D and E.

a parallel CD version, and its performance results are shown in Figure 16. For comparison, we also integrate the technique of coarse-grained parallelism into gzip. With coarse-grained parallelism, both gzip and CD gain performance benefits, and the average performance speedup of CD reaches 2.4X.

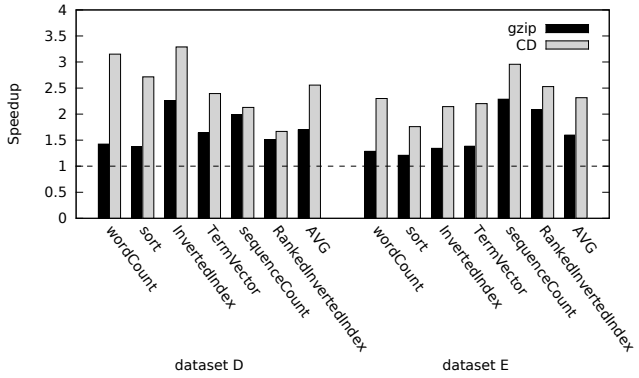


Fig. 16: Parallel performance of different methods normalized to the baseline method on the Single Node machine.

*7.2.2 Time Breakdowns*

The right eight columns in Table 3 report the time breakdowns on datasets D and C, the smallest and the largest ones. Execution on D happens on the Single-Node server and that on C on the Spark Cluster. The time breakdown shows that CD experiences a much shorter I/O time than gzip does. This is because CD needs to load only the compressed data into memory while gzip needs to load the decompressed data. Although the data loading time of **CD** is shorter than that of the **gzip** method, the data loading time only occupies a small proportion of the whole execution time. The major benefits of TADOC come from the effective data reuse.

Even if I/O time is not counted, CD still outperforms gzip substantially. This is reflected in CD's shorter times in all other parts of the time breakdowns. For instance, CD's initialization step takes about 1/3 to 1/2 of that of gzip. This is because gzip requires significant time to produce the completely decompressed data.

In most cases, the actual data processing part of CD (i.e., the "compute time" column in Table 3) is also much shorter than that of gzip, thanks to CD's avoidance of the repeated processing of content that appears *multiple* times in the input datasets.[2] The exceptions are `sequence count` and `ranked inverted index` on dataset D. These two programs are both unit and order sensitive. Dataset D, which consists of many small files, does *not* have many repeated word sequences, so obtaining performance improvement on it is even harder. However, even for these two extreme cases, the overall time of CD is still shorter than that of gzip because of CD's substantial savings in the I/O and initialization steps. We conclude that our CD method greatly reduces execution time on many workloads and datasets.

[2] The processing time in gzip is the same as in the baseline method since they both process the decompressed data.

## 7.3 Space Savings

Table 4 reports the compression ratio, which is defined as *size(original)/size(compressed)*. In all methods that use compression, the datasets are already dictionary-encoded. Compression methods apply to both the datasets and the dictionary. The CD- row shows the compression ratios from Sequitur alone. Sequitur's compression ratio is 2.3–3.8, considerably smaller than the ratios from Gzip. However, with the double compression technique, CD's compression ratio is boosted to 6.5–14.1, which is greater than the Gzip ratios. Gzip results *cannot* be used for direct data processing, but Sequitur results can, which enables CD to bring significant time savings as well, as reported in Section 7.2.

The "Memory" columns in Table 3 report the memory savings by "CD" compared to the memory usage by the gzip method. Because CD loads and processes much less data, it reduces memory usage by 87.9%. This benefit is valuable considering the large pressure modern analytics pose to the memory space of modern machines. The smaller memory footprint also helps CD to reduce memory access times.

## 7.4 Evaluation of Advanced Document Analytics

*7.4.1 Time Benefits*

In this section, we measure the performance of TADOC in the four applications described in Section 6. We show the execution time benefits in Figure 17. As introduced in Section 7.1, the baseline is the execution time of the original implementation, which directly processes the original non-compressed dataset. gzip uses Gzip to compress the original data, and during processing time, it needs to decompress the compressed data before data processing. CD is the version using TADOC, which also includes a gunzip stage. We use the speedup over the baseline as the metric to quantify time benefits. The gzip version suffers from data decompression overheads, so we focus on the analysis of the CD version of TADOC.

In Figure 17, CD yields 1.2X speedup on average over the baseline. Among the four applications, TFIDF experiences a performance benefit greater than 30%. The reason for this result is that these applications can be divided into two stages, namely, data preprocessing and

Table 3: Time breakdown (seconds) and memory savings.

| | | Memory | | I/O Time | | Init Time | | Compute Time | | Total Time | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Benchmark | gzip (MB) | CD savings (%) | gzip | CD | gzip | CD | gzip | CD | gzip | CD |
| | word count | 1157.0 | 88.8 | 4.0 | 2.6 | 14.1 | 4.5 | 0.4 | 0.3 | 18.5 | 7.4 |
| dataset D | sort | 1143.0 | 88.7 | 4.0 | 2.6 | 15.0 | 6.1 | 0.4 | 0.3 | 19.4 | 8.9 |
| data size: 0.9 GB | inverted index | 1264.7 | 79.5 | 4.0 | 2.6 | 13.4 | 4.0 | 11.1 | 6.2 | 28.5 | 12.8 |
| CD size: 132 MB | term vector | 1272.1 | 71.0 | 4.0 | 2.6 | 13.3 | 7.4 | 4.1 | 3.3 | 21.4 | 13.2 |
| storage saving: 84.7% | sequence count | 1734.3 | 47.3 | 4.0 | 2.6 | 13.8 | 4.1 | 50.4 | 58.3 | 68.1 | 65.0 |
| | ranked inverted index | 1734.3 | 47.3 | 4.0 | 2.6 | 13.9 | 4.4 | 138.7 | 141.5 | 156.6 | 148.4 |
| | word count | 177920.0 | 89.5 | 571.5 | 131.5 | 3120.0 | 840.0 | 900.0 | 780.0 | 4591.5 | 1751.5 |
| dataset C | sort | 177920.0 | 89.5 | 511.5 | 131.5 | 2940.0 | 780.0 | 1500.0 | 1200.0 | 4951.5 | 2111.5 |
| data size: 144.4 GB | inverted index | 180638.0 | 88.1 | 596.1 | 120.0 | 4140.0 | 600.0 | 1380.0 | 480.0 | 6116.1 | 1200.0 |
| CD size: 11 GB | term vector | 184138.0 | 86.5 | 571.5 | 131.5 | 3540.0 | 660.0 | 1560.0 | 780.0 | 5671.5 | 1571.5 |
| storage saving: 92.4% | sequence count | 205117.8 | 77.6 | 672.9 | 320.0 | 5820.0 | 3780.0 | 1380.0 | 1500.0 | 7872.9 | 5600.0 |
| | ranked inverted index | 205117.8 | 77.6 | 672.9 | 260.0 | 7020.0 | 5280.0 | 3600.0 | 3480.0 | 11292.9 | 9020.0 |

Table 4: Compression ratios.

| | Dataset | | | | | |
|---|---|---|---|---|---|---|
| Version | A | B | C | D | E | AVG |
| default | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| gzip | 9.3 | 8.9 | 8.5 | 5.9 | 8.9 | 8.3 |
| CD | 14.1 | 13.3 | 13.1 | 6.5 | 11.9 | 11.8 |
| CD- | 3.1 | 3.2 | 3.8 | 2.3 | 2.8 | 3.0 |

CD-: Sequitur without double compression.

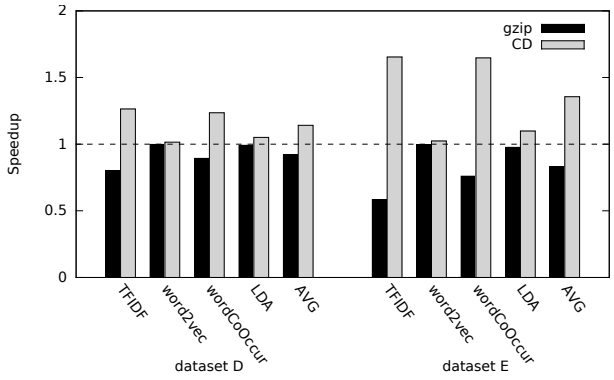


Fig. 17: Performance of different methods normalized to the baseline of the advanced document analytics applications.

processing, and TADOC is mainly used in the data preprocessing stage; the preprocessing stage may account for different proportions of the total execution time. The time breakdown is shown in Table 5, where the last column represents the proportion of preprocessing time. Data preprocessing accounts for 44.6% (`D`) and 64.7% (`E`) of execution time for TFIDF. For word co-occurrence, data preprocessing accounts for 35.4% (`D`) and 21.3% (`E`) of execution time. The preprocessing proportion of these two applications is high compared to that of the other applications, and additionally, the decompression process of `CD` is shorter than that of `gzip`. Therefore, TFIDF and word co-occurrence have relatively high performance speedups.

In general, when the data preprocessing and decompression stages account for a high proportion of the total execution time, TADOC has significant performance benefits; otherwise, the execution time of TADOC is lower than but closer to the original execution time.

Table 5: Time breakdown for advanced applications.

| | Benchmark | Preprocessing (s) | Total Time (s) | Occupancy (%) |
|---|---|---|---|---|
| D | TFIDF | 5.6 | 12.6 | 44.6 |
| | wordCoOccur | 9.6 | 27.2 | 35.4 |
| | LDA | 20.3 | 448.9 | 4.5 |
| | word2vec | 7.0 | 2082.6 | 0.3 |
| E | TFIDF | 7.4 | 11.5 | 64.7 |
| | wordCoOccur | 5.5 | 25.9 | 21.3 |
| | LDA | 14.9 | 502.9 | 3.0 |
| | word2vec | 7.0 | 2864.0 | 0.2 |

#### *7.4.2 Storage and Memory Benefits*

We evaluate the storage and memory benefits of TADOC on advanced analytics workloads. Storage savings are the same as those in the results in Section 7.3. Compared to the original datasets, `CD` reaches 6.5 and 11.9 compression ratios for datasets `D` and `E`, respectively, which implies that TADOC brings more than 90% storage reduction.

Memory savings are shown in Table 6 and range from 0.6% to 76.2%. They vary considerably because different applications use different auxiliary data structures. For example, for TFIDF, the algorithm is simple, and we only need to develop data structures for storing the TF and IDF. In contrast, for LDA, although the "bag-of-words" paradigm used in preprocess relies on only word frequencies, the training stage involves many intermediate data structures in model construction, which decreases TADOC's memory benefits.

Table 6: Memory savings for advanced applications.

| | Benchmark | Original (MB) | Memory Savings (%) |
|---|---|---|---|
| D | TFIDF | 1617.0 | 60.5 |
| | wordCoOccur | 1679.0 | 17.7 |
| | LDA | 2552.4 | 0.6 |
| | word2vec | 772.3 | 17.1 |
| E | TFIDF | 1459.4 | 76.2 |
| | wordCoOccur | 1881.0 | 15.2 |
| | LDA | 3883.6 | 3.4 |
| | word2vec | 911.0 | 18.4 |

### 7.5 When Inverted Index is Used

In some cases, practitioners store an inverted index [95, 90, 59] with the original dataset. Doing so helps accelerate some analytics tasks. This approach can be combined with TADOC by attaching an inverted index of

the original documents with the Sequitur compression result. We call these two schemes Original+index and CD+index. For tasks where inverted index can be used (e.g., the first four benchmarks), some intermediate results can be obtained directly from inverted index to save time. For the other tasks (e.g., `sequence count`, `ranked inverted index`), Original+index has to fall back to the original text for analysis, and CD+index provides 1.2X-1.6X speedup due to its direct processing on the Sequitur DAG. Besides its performance benefits, CD+index saves about 90% space over the Original+index as Table 7 reports.

Table 7: Space usage of the original datasets and CD with inverted-index.

| Usage | Dataset | A | B | C | D | E |
|---|---|---|---|---|---|---|
| Memory | Original+Index | 32,455 | 92,234 | 184,469 | 1,387 | 1,406 |
| (MB) | CD+Index | 3,693 | 10,405 | 20,806 | 413 | 197 |
| Storage | Original+Index | 37,990 | 78,438 | 154,214 | 1,115 | 1,559 |
| (MB) | CD+Index | 2,873 | 6,066 | 11,965 | 211 | 140 |

### 7.6 More Detailed Benefits

In this part, we briefly report the benefits coming from each of the major guidelines we described in Section 4.

The benefits of *adaptive traversal order* (Guideline I and II) are most prominent on benchmarks `inverted index` and `term vector`. Using adaptive traversal order, the `CD` method selects postorder traversal when processing dataset `D` and preorder on datasets `A, B, C, E`. We show the performance of both preorder and postorder traversals for `inverted index` and `term vector` in Figure 18. Using decision trees, `CD` successfully selects the better traversal order for each of the datasets. For instance, on `inverted index`, `CD` picks postorder on dataset `D`, which outperforms preorder by 1.6X, and it picks preorder on dataset `E`, which outperforms postorder by 1.3X.

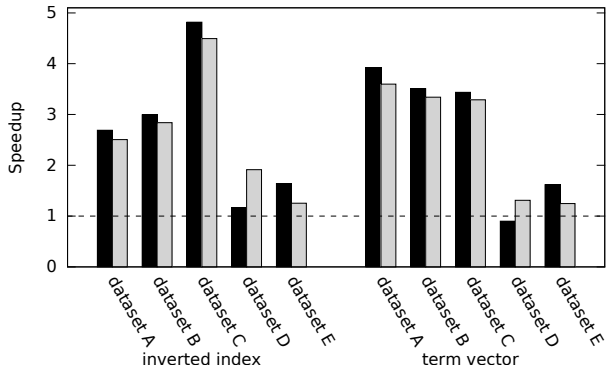


Fig. 18: Performance of preorder and postorder for `inverted index` and `term vector`.

*Double compression* (Guideline VI) provides substantial space benefits as we have discussed in Section 4.5. However, since double compression needs to recover the Sequitur results from the compressed data before processing, it incurs some overhead. Our experiments show that this overhead is outweighed by the overall time benefits of `CD`.

We tried to implement a fine-grained parallel version of `CD` for benchmark `word count`. It breaks the CFG into a number of partitions and uses multiple threads to process each partition in parallel. Even though this version took us several times the effort we spent on the *coarse-grained parallel* version (Guideline III), its performance was substantially worse (e.g., 50% slower on dataset `D`).

*Double-layered bitmap* (Guideline V) helps *preorder* processing on datasets that contain many (>800) files of medium size (>2860 words per Figure 6.) Among the 60 datasets involved in the decision tree experiments in Section 4.2, 10 of them works best with double-layered bitmap based preorder. They get 2%-10% performance benefits compared to single-layered bitmap based preorder traversal. Besides double-layered bitmap, we experiment with other alternative data structures, including *red-black tree* [23], *hash set* [23], and *B-tree* [4]. Table 8 reports the performance of preorder `inverted index` when these data structures are used in place of double-layered bitmap in each of the DAG node. The experiment uses dataset D and the Single-Node server in Table 2. Double-layered bitmap is fast to construct as it uses mainly bit operations. The query time for double-layered bitmap has a complexity of $O(1)$. Some of the alternative data structures (e.g., B-tree) yield a shorter processing time, but suffer a longer construction process (i.e., initialization in Table 8).

Table 8: Performance and time breakdown of different data structure achieves for inverted-index.

| Data Structure | Initialization (s) | Computation (s) | Total (s) |
|---|---|---|---|
| 2LevBitMap | 14.96 | 3.08 | **18.04** |
| redBlackTree | 39.33 | 3.56 | **42.89** |
| hash set | 25.34 | 4.32 | **29.67** |
| B-tree | 18.87 | 2.29 | **21.16** |

Finally, *coarsening* (Guideline VI) shows clear benefits for `CD` on benchmarks `ranked inverted index` and `sequence count`. For instance, compared to no coarsening, it enables the CD-based `ranked inverted index` program to achieve 5% extra performance improvement on dataset `E`. The benefits of Guideline IV has been reported in Section 4.4 and are hence omitted here.

### 7.7 Compression Time and Applicability

The time taken to compress the datasets using sequential Sequitur ranges from 10 minutes to over 20 hours. Using a parallel or distributed Sequitur with accelerators (e.g., as in [14,70]) can potentially shorten the compression time substantially. Note that this article

focuses on how to use the compressed data to support various analytics tasks such as word count; we do not further discuss the compression process.

In general, our technique is designed for document analytics that can be expressed as a DAG traversal-based problem on datasets that do *not* change frequently. Moreover, our discussion has focused on applications that normally require scanning the entire dataset, as illustrated by the analytics problems used in our evaluation. It is not designed for regular expression queries or scenarios where data frequently changes. We note that the proposed technique can also benefit advanced document analytics. The initial part of many advanced document analytics is to load documents and derive some representations (e.g., natural language understanding) to characterize the documents such that later processing can efficiently work on these representations. One example is email classification, where TADOC can help to accelerate the construction of the feature vectors (e.g., word frequency vectors) required for classification. For the applications that cannot be expressed as a DAG traversal-based problem, TADOC can be used as a storage technique. For example, to compute Levenshtein distance between a pair of words [50], the required words can be extracted from the compressed data [99], and then, the application of Levenshtein edit distance can be performed on the extracted words.

## 8 Related Work

To our knowledge, this is the first work to enable *efficient* direct document analytics on compressed data. The work closest to `CompressDirect` is Succinct [8, 42], which enables efficient queries on compressed data in a database. These two techniques are complementary to each other. Succinct is based on index and suffix array [62], an approach employed in other works as well [8,17,29,37,25]. `CompressDirect` and these previous studies differ in both their applicability and main techniques. First, Succinct is mainly for the database domain while `CompressDirect` is for general document analytics. Succinct is designed mainly for search and random access of local queries. Even though it could possibly be made to work on some of the general document analytics tasks, its efficiency is much less than `CompressDirect` on such tasks as those tasks are not its main targets. For instance, `word count` on dataset `E` takes about 230 seconds with Succinct, but only 10.3 seconds with `CompressDirect`, on the single node machine in Table 2. Second, Succinct and `CompressDirect` use different compression methods and employ different inner storage structures. Succinct compresses data in a flat manner, while `CompressDirect` uses Sequitur to create a DAG-like storage structure. The DAG-like structure allows `CompressDirect` to efficiently perform general computations for all items in the documents, even in the presence of various challenges from files or word sequences, as described in Section 3.2.

Note that Sequitur can be replaced by other context-free compression techniques. For example, Re-Pair, proposed by Larsson and Moffat [45], is an offline dictionary-based compression algorithm. Re-Pair can be regarded as a compromise in terms of compression time and compression ratio, which could be a potential alternative compression algorithm. Larson and Moffat [2] offer a high-quality Re-Pair implementation. Gańczorz and others [33] further improve the Re-Pair grammar compressor by involving penalties.

Traditional approaches to compression-based analytics use indexes, suffix arrays, and suffix trees [62,8, 17,29,37,25,24,30,25,34]. Suffix trees [62,84] are traditional compact data structures; they consume less storage space while enabling analytics on compressed data. However, research [39,44] shows that optimized representations consume larger memory even more than the size of the input. Burrows-Wheeler Transform [29,17] and suffix arrays [56] are milestones in the development of compact representations, but experiments [39] still show that they cannot solve the large memory consumption issue. FM-indexes [6,26–29] and Compressed Suffix Array [36,38,81–83] are two efficient alternatives that further reduce the memory space consumption, and, based on these technologies, Agarwal and others propose Succinct [8]. Our method, TADOC, is different from these methods; we provide a grammar-based approach, which provides new insight to the domain of compression-based data analytics.

There are many works on grammar compression and operations on grammar-encoded strings [80,19,31,11, 10,16,32,87]. Rytter [80] survey the complexity issues involved in grammar compression, LZ-Encodings, and string algorithms. Charikar and others [19] study the limit of the smallest context-free grammar to generate a given string and analyze the bound approximation ratios for well-known grammar-based compression algorithms, including Sequitur. Gagie and others [31] develop a novel grammar-based self-indexing method for highly-repetitive strings such as DNA sequences. Bille and others [11] study how to perform random access to grammar-compressed data. SOLCA [87] is a novel fully-online grammar compression algorithm that targets online use cases.

There is a large body of literature on inverted index compression. Petri and Moffat [73] explore general-purpose compression tools for compact inverted index storage. Moffat and Petri [60] apply two-dimensional contexts and the byte-aligned entropy coding of asym-

metric numeral systems to index compression. Pibiri and others [76] develop a fast dictionary-based compression approach, DINT, for inverted indexes. Pibiri and Venturini [77] survey the encoding algorithms for inverted index, including single integers, a sorted list of integers, and the inverted index. Pibiri and others [75] propose compressed indexes for fast searching of large RDF datasets. Oosterhuis and others [68] apply the recursive graph bisection in document reordering, which is an essential preprocessing phase in building indexes. Furthermore, Mackenzie and others [54] use machine learned models to replace common index data structures. These works mainly consider how to compress the inverted index. Different from inverted index compression, TADOC focuses on how to perform analytics directly on compressed data without decompression, such as building inverted indexes directly on compressed data.

Since its development [64–66], Sequitur has been applied to various tasks, including program and data pattern analysis [20,21,46–48,52,89]. Lau and others [47] use Sequitur in code analysis to search some program patterns. Chilimbi [20] uses Sequitur as a representation to quantify and exploit data reference locality for program optimization. Larus [46] proposes an approach called whole program paths (WPP), which leverages Sequitur to capture and represent dynamically executed control flow. Law and others [48] propose a whole program path-based dynamic impact analysis and related compression based on Sequitur. Chilimbi and others [21] use Sequitur for fast detection of hot data streams. Walkinshaw and others [89] apply Sequitur to the comprehension of program traces at varying levels of abstraction. Lin and others [52] extend Sequitur as a new XML compression scheme for supporting query processing. We are not aware of prior usage of Sequitur to support direct document analytics on compressed data, as we propose.

## 9 Conclusion

We propose a new method, TADOC, to enable high performance document analytics on compressed data. By enabling efficient direct processing on compressed data, our method reduces storage space by 90.8% and memory usage by 87.9%, while also speeding up the analytics by 1.6X on sequential systems, and 2.2X on distributed clusters. We present how the proposed method can be materialized on Sequitur, a compression method that produces hierarchical grammar-like representations. We discuss the major challenges in applying the method to various document analytics tasks, and provide a set of guidelines for developers to avoid potential pitfalls in applying TADOC. In addition, we produce a library named `CompressDirect` to help ease the required development effort in using TADOC. Our results demonstrate the promise of TADOC in various environments, ranging from sequential to parallel and distributed systems.

**Acknowledgements** This work is supported by the National Key R&D Program of China (Grant No. 2017YFB1003103), National Natural Science Foundation of China (No. 61732014 and 61802412) Beijing Natural Science Foundation (No. 4202031 and L192027), Tsinghua University Initiative Scientific Research Program (20191080594), and Beijing Academy of Artificial Intelligence (BAAI). Onur Mutlu is supported by ETH Zürich, SRC, and various industrial partners of the SAFARI Research Group, including Alibaba, Huawei, Intel, Microsoft, and VMware. Jidong Zhai, Xipeng Shen, and Xiaoyong Du are the corresponding authors of this paper.

## References

1. Amazon elastic compute cloud (Amazon EC2). `https://aws.amazon.com/ec2/`.
2. Re-Pair compression and decompression. `https://users.dcc.uchile.cl/~gnavarro/software/index.html`, 2010.
3. word2vec. https://code.google.com/archive/p/word2vec/, 2013.
4. C++ B-tree. https://code.google.com/archive/p/cpp-btree/, 2017.
5. Wikipedia HTML data dumps. https://dumps.wikimedia.org/enwiki/, 2017.
6. FM-index. https://en.wikipedia.org/wiki/FM-index, 2018.
7. zstd. https://facebook.github.io/zstd/, 2020.
8. R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, 2015.
9. F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. PUMA: Purdue MapReduce Benchmarks Suite. 2012.
10. P. Bille, A. R. Christiansen, P. H. Cording, and I. L. Gørtz. Finger search in grammar-compressed strings. *arXiv preprint arXiv:1507.02853*, 2015.
11. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 2015.
12. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 2003.
13. J. E. Blumenstock. Size matters: word count as a measure of quality on Wikipedia. In *WWW*, 2008.
14. A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.
15. D. Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfs design. pdf*, 2008.
16. N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. R. Paramá. Gract: a grammar-based compressed index for trajectory data. *Information Sciences*, 2019.
17. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.
18. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

19. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 2005.
20. T. M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *PLDI*, 2001.
21. T. M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-purpose Programs. In *PLDI*, 2002.
22. J. P. Chiu and E. Nichols. Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the Association for Computational Linguistics*, 2016.
23. T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.
24. A. Farruggia, P. Ferragina, and R. Venturini. Bicriteria data compression: efficient and usable. In *European Symposium on Algorithms*, 2014.
25. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 2009.
26. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000.
27. P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences*, 2001.
28. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, 2001.
29. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 2005.
30. P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of lempel-ziv compression. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009.
31. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *International Conference on Language and Automata Theory and Applications*, 2012.
32. M. Ganardi, A. Jeż, and M. Lohrey. Balancing straight-line programs. In *IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, 2019.
33. M. Gańczorz and A. Jeż. Improvements on Re-Pair grammar compressor. In *Data Compression Conference (DCC)*, 2017.
34. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, 2014.
35. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
36. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.
37. R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004.
38. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 2005.
39. W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences. In *ALENEX/ANALC*, 2004.
40. S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *New Frontiers in Information and Software as Services*. 2011.
41. T. Joachims. A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization. Technical report, Carnegie-mellon univ pittsburgh pa dept of computer science, 1996.
42. A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.
43. T. Koiwa and H. Ohwada. Extraction of disease-related genes from PubMed paper using word2vec. In *Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics*, 2017.
44. S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 1999.
45. N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 2000.
46. J. R. Larus. Whole Program Paths. In *PLDI*, 1999.
47. J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *International Symposium on Performance Analysis of Systems and Software*, 2005.
48. J. Law and G. Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *ICSE*, 2003.
49. L. Lebart. Classification problems in text analysis and information retrieval. In *Advances in Data Science and Classification*. 1998.
50. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966.
51. M. Lichman. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2013.
52. Y. Lin, Y. Zhang, Q. Li, and J. Yang. Supporting efficient query processing on compressed XML files. In *Proceedings of ACM symposium on Applied Computing*, 2005.
53. Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing. *ACM Trans. Intell. Syst. Technol.*, 2011.
54. J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *European Conference on Information Retrieval*, 2019.
55. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
56. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 1993.
57. C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg. *Practical Graph Analytics with Apache Giraph*. Springer, 2015.
58. Y. Matsuo and M. Ishizuka. Keyword extraction from a single document using word co-occurrence statistical information. *International Journal on Artificial Intelligence Tools*, 2004.
59. K. Mitsui. Information retrieval based on rank-ordered cumulative query scores calculated from weights of all keywords in an inverted index file for minimizing access to a main database, 1993. US Patent 5,263,159.

60. A. Moffat and M. Petri. Index compression using byte-aligned ANS coding and two-dimensional contexts. In *WSDM*, 2018.
61. A. E. Monge, C. Elkan, et al. The Field Matching Problem: Algorithms and Applications. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1996.
62. G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
63. C. G. Nevill-Manning. *Inferring sequential structure*. PhD thesis, University of Waikato, 1996.
64. C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 1997.
65. C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 1997.
66. C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference*, 1997.
67. B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.
68. H. Oosterhuis, J. S. Culpepper, and M. de Rijke. The potential of learned index structures for index compression. In *Proceedings of the 23rd Australasian Document Computing Symposium*, 2018.
69. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 2011.
70. G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *MICRO*, 2013.
71. J. W. Pennebaker, M. E. Francis, and R. J. Booth. Linguistic inquiry and word count: LIWC 2001. *Mahway: Lawrence Erlbaum Associates*, 2001.
72. J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.
73. M. Petri and A. Moffat. Compact inverted index storage using general-purpose compression libraries. *Software: Practice and Experience*, 2018.
74. F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *CIKM*, 2015.
75. G. E. Pibiri, R. Perego, and R. Venturini. Compressed Indexes for Fast Search of Semantic Data. *TKDE*, 2020.
76. G. E. Pibiri, M. Petri, and A. Moffat. Fast dictionary-based compression for inverted indexes. In *WSDM*, 2019.
77. G. E. Pibiri and R. Venturini. Techniques for Inverted Index Compression. *arXiv preprint arXiv:1908.10598*, 2019.
78. I. Popov. Malware detection using machine learning based on word2vec embeddings of machine code instructions. In *2017 Siberian Symposium on Data Science and Engineering (SSDSE)*, 2017.
79. X. Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.
80. W. Rytter. Grammar compression, lz-encodings, and string algorithms with implicit input. In *International Colloquium on Automata, Languages, and Programming*, 2004.
81. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *International Symposium on Algorithms and Computation*, 2000.
82. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002.
83. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 2003.
84. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2007.
85. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of discrete Algorithms*, 2007.
86. M. Sharma. Compression using Huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 2010.
87. Y. Takabatake, H. Sakamoto, et al. A space-optimal grammar compression. In *25th Annual European Symposium on Algorithms*, 2017.
88. F. Vasile, E. Smirnova, and A. Conneau. Meta-prod2vec: Product embeddings using side-information for recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016.
89. N. Walkinshaw, S. Afshan, and P. McMinn. Using compression algorithms to support the comprehension of program traces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*, 2010.
90. K.-Y. Whang, B.-K. Park, W.-S. Han, and Y.-K. Lee. Inverted index storage structure using subindexes and large objects for tight coupling of information retrieval with database management systems, 2002. US Patent 6,349,308.
91. R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, 2013.
92. A. Xu, Z. Liu, Y. Guo, V. Sinha, and R. Akkiraju. A new chatbot for customer service on social media. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017.
93. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.
94. U. Zernik. *Lexical acquisition: exploiting on-line resources to build a lexicon*. Psychology Press, 1991.
95. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD*, 2001.
96. F. Zhang, B. Wu, J. Zhai, B. He, W. Chen, and X. Du. Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. *TKDE*, 2019.
97. F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *PVLDB*, 2018.
98. F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen. Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data. In *ICS*, 2018.
99. F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du. Enabling efficient random access to hierarchically-compressed data. In *ICDE*, 2020.
100. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.
101. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 2006.