Efficient Document Analytics on Compressed Data: Method, Challenges, Algorithms, Insights

Feng Zhang [†‡⋄], Jidong Zhai [⋄], Xipeng Shen [#], Onur Mutlu [⋆], Wenguang Chen [⋄]
[†] Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, China
[‡] School of Information, Renmin University of China
[⋄] Department of Computer Science and Technology, Tsinghua University, China
[#] Computer Science Department, North Carolina State University, USA
[⋆] Department of Computer Science, ETH Zürich, Switzerland
fengzhang@ruc.edu.cn zhaijidong@tsinghua.edu.cn xshen5@ncsu.edu
onur.mutlu@inf.ethz.ch cwg@tsinghua.edu.cn

ABSTRACT

Today's rapidly growing document volumes pose pressing challenges to modern document analytics, in both space usage and processing time. In this work, we propose the concept of *compression-based direct processing* to alleviate issues in both dimensions. The main idea is to enable direct document analytics on compressed data. We present how the concept can be materialized on Sequitur, a compression algorithm that produces hierarchical grammar-like representations. We discuss the major challenges in applying the idea to various document analytics tasks, and reveal a set of guidelines and also assistant software modules for developers to effectively apply *compression-based direct processing*. Experiments show that our proposed techniques save 90.8% storage space and 77.5% memory usage, while speeding up data processing significantly, i.e., by 1.6X on sequential systems, and 2.2X on distributed clusters, on average.

PVLDB Reference Format:
Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, Wenguang Chen. Efficient Document Analytics on Compressed Data: Method, Challenges, Algorithms, Insights. *PVLDB,* 11(11): 1522-1535, 2018.
DOI: https://doi.org/10.14778/3236187.3236203

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 [2]) 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 paper presents our systematic exploration of document analytics on compressed data. We base our study on a specific compression algorithm named *Sequitur* [35] 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.

Our evaluation validates the efficacy of our proposed techniques for saving both space and time. Compared to data analytics on the original uncompressed datasets, our tech-

Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07.
DOI: https://doi.org/10.14778/3236187.3236203

niques reduce storage usage by 90.8% and memory usage by 77.5%. At the same time, they speed up the analytics by 1.6X for sequential runs, and by 2.2X for Spark-based distributed runs.

A prior work, Succinct [3], 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 [30] saves storage space, but does *not* save repeated processing of the data.

Overall, this work makes the following contributions:

- It presents the first method for enabling high performance complex document analytics directly on compressed data, and realizes the method on *Sequitur*.
- 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.

2. PREMISES AND BACKGROUND

2.1 Sequitur Algorithm

There are many compression algorithms for documents, such as LZ77 [53], suffix array [32], and their variants. Our study focuses on Sequitur [35] 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.

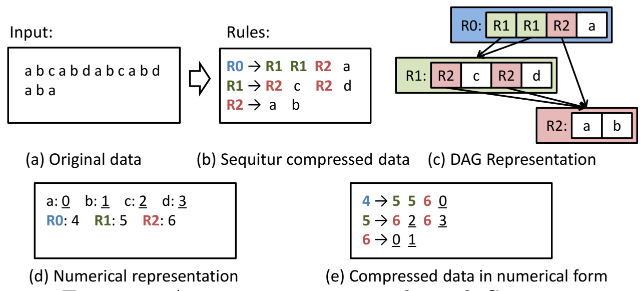


Figure 1: A compression example with Sequitur.

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 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) [33]. 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 [2], 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 [53] and suffix array [32]). 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* [5, 39, 4] 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* [4, 13, 47] builds a word-to-file index for a document dataset. It is widely used in search engines. The input of *inverted index* is a set of files, and the output is the mappings between words and files. Unlike *word count*, *inverted index* distinguishes between files on its output.

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

Sequence Count *Sequence count* [4, 49, 24] 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. COMPRESSION-BASED DIRECT PROCESSING AND ITS CHALLENGES

In this section, we present the concept of *compression-based direct processing*, including its basic algorithms and the challenges for materializing it effectively.

3.1 Compression-Based Direct Processing

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.

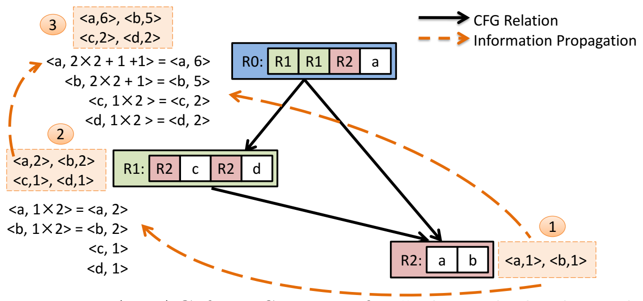


Figure 2: A DAG from Sequitur for "a b c a b d a b c a b d a b a", and a postorder traversal of the DAG for counting word frequencies.

Because the processing leverages the compression results, it naturally avoids repeated processing of repeated content. The repeated content needs to be counted 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 example illustrates the essence of the general algorithm of our *compression-based direct processing* method:

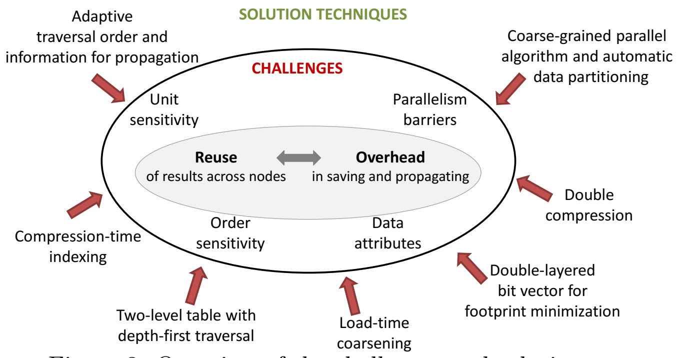
```
Let G be the graph representing Sequitur compression results.
Conduct a traversal of G, during which, at each node, do the following:
(1) Local Processing:
    Process local info;
(2) Data Integration:
    Integrate the local info with results passed to this node during the traversal;
(3) Data Propagation:
    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, compression-based direct processing 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 *compression-based direct processing* on document analytics 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 *compression-based direct processing* is to maximize the reuse while minimizing the overhead.



SOLUTION TECHNIQUES
Adaptive traversal order and information for propagation
Coarse-grained parallel algorithm and automatic data partitioning
CHALLENGES
Unit sensitivity
Parallelism barriers
Reuse of results across nodes
Overhead in saving and propagating
Order sensitivity
Data attributes
Double compression
Compression-time indexing
Double-layered bit vector for footprint minimization
Two-level table with depth-first traversal
Load-time coarsening

Figure 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 *compression-based direct processing*. 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, *compression-based direct processing* 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 *compression-based direct processing.*

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 software 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 *compression-based direct processing.* 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, we use *inverted index* as an example to explain our insights. We also describe some basic ideas for handling unit sensitivity.

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 4 illustrate. (This usage of special markers offers a general way to handle unit sensitivity.)

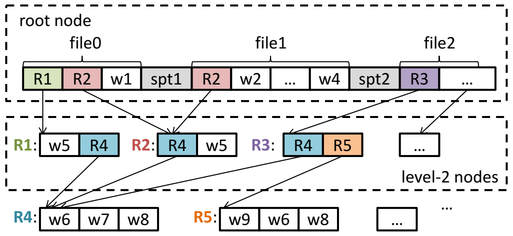


Figure 4: Sequitur compression result with file separators ("spt1" and "spt2" are two file separators).

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 4, 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.

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.

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 6.1, *NSF Research Award Abstracts* dataset [25]), 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 (see [51] for details).

Guidelines and Software Module

Our experience leads to the following two guidelines for implementing *compression-based direct processing.*

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 [38]) 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 5 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 6. 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.

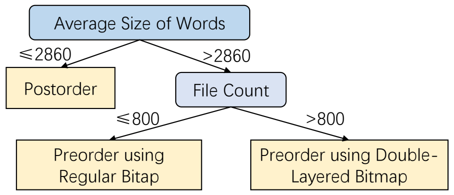

Average Size of Words
⩽2860
>2860
Postorder
File Count
⩽800
>800
Preorder using Regular Bitap
Preorder using Double-Layered Bitmap

Figure 5: Decision tree for choosing traversal order.

4.3 Coarse-Grained Parallelism and Data Partitioning

To obtain scalable performance, it is important for *compression-based direct processing* 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 [17, 40, 9, 28, 27, 46]. For instance, PowerGraph [17], exploits the power-law property of graphs for distributed graph placement and representation, and HDRF [40] 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. 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 1 shows the pseudo-code. The depth-first graph traversal is embodied by the recursive function `seqCount` (lines 10 and 15 in Algorithm 1). 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 then this newly formed sequence in `q` is processed, resulting in the incrementing of 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 has been built such that rebuildings can be avoided.

Figure 6 demonstrates how Algorithm 1 works on an input word sequence whose DAG is shown in Figure 4. 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

Algorithm 1 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 4 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:     $rule.locTblReady = true$
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
```

in the global table. The bottom sequence ⑤ is the same as the second sequence ②; the algorithm does *not* recount this sequence, but directly reuses the entry in the local table of R4.

The key for the correctness of Algorithm 1 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 1 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 1 ensures the correct visiting order of the words it contains. The correctness hence follows.

Lemma 3: The depth-first traversal by Algorithm 1 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

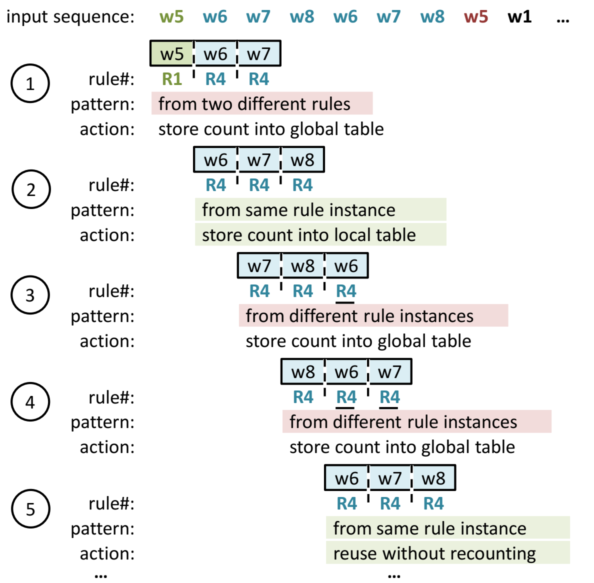


Figure 6: Illustration of how Algorithm 1 processes an input sequence (DAG in Figure 4) for counting 3-word long sequences.

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 1. 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 1 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 1 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 1 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 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.

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

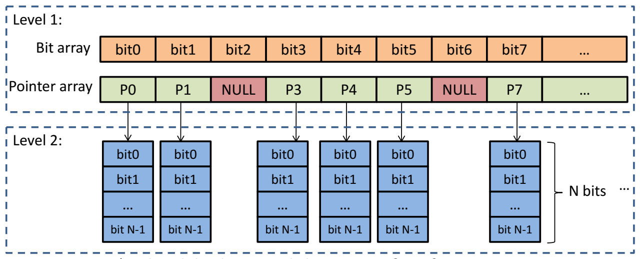
We introduce three extra optimizations. They are mostly implementation-level features that are useful for deploying compression-based direct processing efficiently.

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 is a data structure that has been used in many tasks. Our contribution is in recognizing its benefits for compression-based direct processing.

Recall that in the preorder design of *inverted index* in Section 4.2, we need to propagate file sets across nodes. One possible design is that each node uses a *set* structure to store file IDs. However, frequent querying of and insertions to the set limit the performance of graph traversal. An alternative is that each node uses a *bit-vector*, with each bit corresponding to a file: 1 for file presence, 0 otherwise. Although a bit vector representation converts slow set operations with fast bit operations, it incurs large space overhead when there are many files, as every node needs such a vector, and the length of the vector equals the total number of files.

We find a double-layered bitmap to be effective in addressing the problem. It is inspired by the use of similar data structures in page table indexing of operating systems [42] and indexing mechanisms for graph query [8]. As Figure 7 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 queried. If a rule is contained in only a subset of files, whose bits correspond to some bits in several level-2 vectors, then the level two of the bitmap associated with that rule would 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.



Level 1:
Bit array
bit0 bit1 bit2 bit3 bit4 bit5 bit6 bit7 ...
Pointer array
P0 P1 NULL P3 P4 P5 NULL P7 ...
Level 2:
bit0
bit1
...
bit N-1
N bits
...

Figure 7: 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 5 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 5 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 6).

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 find helpful for the compression step. Compared to some other compression algorithms, our Sequitur-based method is based on words and it does *not* always get the highest compression rates. To solve this issue, we first compress the original dataset with Sequitur and then run "gzip" (or other methods with high compression rates) on the output of Sequitur. 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 *compression-based direct processing*. 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 *compression-based direct processing* 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` [4] counts the number of each word in all of the input documents. `Sort` [20] sorts all the words in the input documents in lexicographic order. `Inverted index` [4] generates a word-to-document index that provides the list of files containing each word. `Term vector` [4] finds the most frequent words in a set of documents. `Sequence count` [4] calculates the frequencies of each three-word sequence in every input file. `Ranked inverted index` [4] 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 [37] for parallelization), and the third is in Scalar on Spark [48]. 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 [52]. We next report the performance of our implementations.

6. 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 [20] and the rest from Puma [4]. 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.

6.1 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 *compression-based direct processing*. 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 [2], used for tests on clusters. `Dataset D` is NSF Research Award Abstracts (NSFRAA) from UCI Machine Learning Repository [25], consisting of a large number (134,631) of small files. `Dataset E` is a collection of web documents downloaded from the Wikipedia database [2], consisting of four large files.

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, and the three large datasets. The cluster is built with an HDFS storage system [6]. 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.

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

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

6.2 Time Savings

6.2.1 Overall Speedups

Figure 8 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.

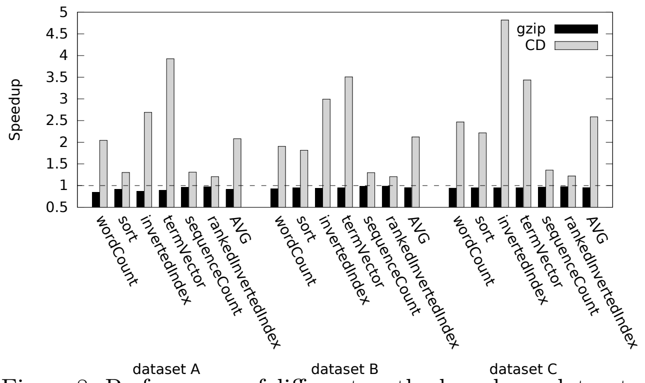


Figure 8: Performance of different methods on large datasets running on the Spark Cluster, normalized to the performance of the baseline method.

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

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

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. Programs **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.

In contrast to the large speedups obtained with the CD method, the **gzip** method provides 1-14% *slowdown* due to the extra decompression time. The decompression time could be partially hidden if a background thread does the decompression while the main thread processes the already-decompressed parts. However, doing so does *not* make the **gzip** method faster than the baseline as the data processing part is still the same as that in the baseline method.

Figure 9 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.

6.2.2 *Time Breakdowns*

The right eight columns in Table 3 report the time breakdowns on datasets D and C, the smallest and the largest

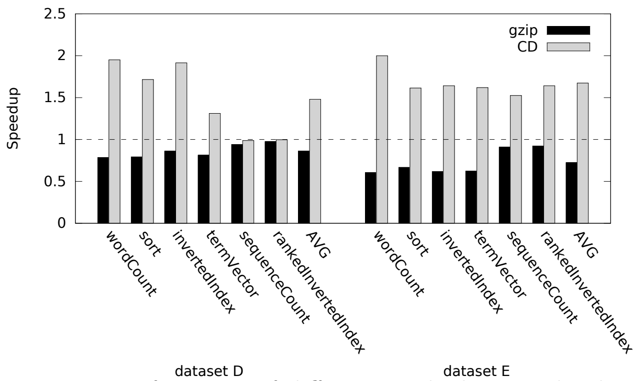


Figure 9: Performance of different methods normalized to the baseline method on the Single Node machine.

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.

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.

6.3 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.

[1]Section 6.5 shows the sensitivity on the other two datasets, D and E.

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

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 6.2.

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.

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 77.5%. 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.

6.4 When Inverted Index is Used

In some cases, practitioners store an inverted index [50, 45, 29] with the original dataset. Doing so helps accelerate some analytics tasks. This approach can be combined with our *compression-based direct processing* 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 5 reports.

Table 5: 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

6.5 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 10. 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.

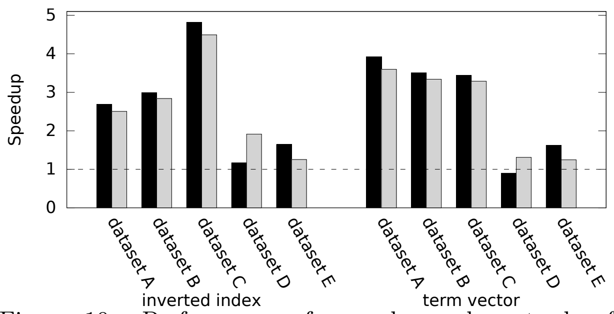


Figure 10: Performance of preorder and postorder for `inverted index` and `term vector`.

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 5.) 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* [12], *hash set* [12], and *B-tree* [1]. Table 6 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 6).

Table 6: 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.

6.6 Compression Time and Applicability

The time to compress the datasets using sequential Sequitur ranges from 10 minutes to over 20 hours. When double compression is employed, Sequitur and Gzip take 94% and 6% of the compression time on average. Using parallel or distributed Sequitur with accelerators can potentially shorten the compression time substantially. We repeat that 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 for many times by different users for various analytics tasks.

Our discussion has focused on applications that normally require scanning the entire dataset. In general, our method is applicable if the text analytics problem can be turned into a DAG traversal-based problem, as illustrated by the six analytics problems used in our evaluation.

Another type of common tasks involve queries that require random accesses to some locations in the dataset. Such tasks are much simpler to support, e.g., by adding some appropriate index to the Sequitur results. Such tasks are already be supported by other recent techniques (e.g., Succinct [3]), and hence are not the focus of our paper.

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. It is *not* designed for regular expression queries or scenarios where data frequently changes. We note that the proposed technique can benefit advanced document analytics as well. 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. An example is email classification, where *compression-based direct processing* can help accelerate the construction of the feature vectors (e.g., word frequency vectors) required for classification.

7. 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 [3], 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 [32], an approach employed in other works as well [3, 7, 16, 18, 15]. `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.

Other prior work tries to enable query processing over encrypted data [43] for security. They do not detect or avoid processing repeated elements in documents. In storage systems, deduplication is a technique used for minimizing the storage of repeated contents [19, 30, 31]. Since deduplication is a technique that works only at the storage level, it does *not* help document analytics to avoid repeatedly processing the content. For space savings, `CompressDirect` and deduplication work at two different computing layers, and are hence complementary to each other, applicable together. When multiple data sources need to be integrated (e.g., in data warehouses), *data cleaning* may detect and remove some redundant data present in different sources [41]. Recent work [14] eliminates redundant computations in document clustering and top-K document retrieval through the application of triangle inequality. However, none of these tasks compress data or support analytics on compressed data.

Since its development [34, 35, 36], Sequitur has been applied to various tasks, including program and data pattern analysis [10, 11, 21, 22, 23, 26, 44]. We are not aware of prior usage of Sequitur to support direct document analytics on compressed data.

8. CONCLUSION

This paper proposes a new method, *compression-based direct processing*, to enable high performance document analytics on compressed data. By enabling efficient direct processing on compressed data, the method saves 90.8% storage space and 77.5% memory usage, while on average, speeding up the analytics by 1.6X on sequential systems, and 2.2X on distributed clusters. The paper presents how the method can be materialized on Sequitur, a compression method that produces hierarchical grammar-like representations. It discusses the major challenges in applying the method to various document analytics tasks, and provides a set of guidelines for developers to avoid potential pitfalls in applying the method. In addition, we produce a library named `CompressDirect` to help ease the required development efforts. Our results demonstrate the promise of the proposed techniques in various environments, ranging from sequential to parallel and distributed systems.

9. ACKNOWLEDGMENTS

This work has been partly supported by the Chinese 863 project 2015AA015306, National Natural Science Foundation of China (Grant No. 61732014, 61722208, 61472201, and 61472427), and Tsinghua University Initiative Scientific Research Program (20151080407). This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. CCF-1455404, CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF. This work is also supported by the China Postdoctoral Science Foundation (2017M620992), and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China (No. 16XNLQ02, 18XNLG07). 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 and Xipeng Shen are the corresponding authors of this paper.

10. REFERENCES

[1] C++ B-tree. https://code.google.com/archive/p/cpp-btree/, 2017.

[2] Wikipedia HTML data dumps. https://dumps.wikimedia.org/enwiki/, 2017.

[3] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[4] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. PUMA: Purdue MapReduce Benchmarks Suite. 2012.

[5] J. E. Blumenstock. Size matters: word count as a measure of quality on Wikipedia. In *Proceedings of the International Conference on World Wide Web*, 2008.

[6] D. Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[8] Q. Q. Cai, H. G. Cui, and H. Tang. Provenance graph query method based on double layer index structure. In *AIP Conference Proceedings*, 2017.

[9] 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.

[10] T. M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[11] T. M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-purpose Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2002.

[12] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[13] D. Cutting and J. Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the International SIGIR Conference on Research and Development in Information Retrieval*, 1990.

[14] Y. Ding, L. Ning, H. Guan, and X. Shen. Generalizations of the Theory and Deployment of Triangular Inequality for Compiler-Based Strength Reduction. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2017.

[15] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 2009.

[16] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 2005.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Symposium on Operating Systems Design and Implementation*, 2012.

[18] 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.

[19] F. Guo and P. Efstathopoulos. Building a High-performance Deduplication System. In *USENIX Annual Technical Conference*, 2011.

[20] 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.

[21] J. R. Larus. Whole Program Paths. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1999.

[22] 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.

[23] J. Law and G. Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the International Conference on Software Engineering*, 2003.

[24] L. Lebart. Classification problems in text analysis and information retrieval. In *Advances in Data Science and Classification*. 1998.

[25] M. Lichman. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2013.

[26] Y. Lin, Y. Zhang, Q. Li, and J. Yang. Supporting efficient query processing on compressed XML files. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[27] 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 *Proceedings of the SIGMOD International Conference on Management of Data*, 2010.

[28] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg. *Practical Graph Analytics with Apache Giraph*. Springer, 2015.

[29] 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.

[30] 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.

[31] G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 2001.

[32] G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[33] C. G. Nevill-Manning. *Inferring sequential structure*. PhD thesis, University of Waikato, 1996.

[34] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 1997.

[35] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 1997.

[36] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference*, 1997.

[37] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[38] 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.

[39] J. W. Pennebaker, M. E. Francis, and R. J. Booth. Linguistic inquiry and word count: LIWC 2001. *Mahway: Lawrence Erlbaum Associates*, 2001.

[40] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *Proceedings of the International on Conference on Information and Knowledge Management*, 2015.

[41] E. Rahm and H. H. Do. Data Cleaning: Problems and Current Approaches. 2000.

[42] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts essentials.* John Wiley & Sons, Inc., 2014.

[43] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[44] 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.

[45] 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.

[46] 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.

[47] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the International Conference on World Wide Web*, 2009.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.

[49] U. Zernik. *Lexical acquisition: exploiting on-line resources to build a lexicon.* Psychology Press, 1991.

[50] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2001.

[51] F. Zhang, J. Zhai, X. Shen, and O. Mutlu. Potential of a Method for Text Analytics Directly on Compressed Data. Technical report, TR-2017-4, Computer Science Department, North Carolina State University, 11 2017.

[52] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen. Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data. In *Proceedings of the International Conference on Supercomputing*, 2018.

[53] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.