Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data

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, China
[⋄]Department of Computer Science and Technology, Tsinghua University
[#]Computer Science Department, North Carolina State University, USA
[⋆]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 frameworks, in both space usage and processing time. Recently, a promising method, called *text analytics directly on compressed data (TADOC)*, was proposed for improving both the time and space efficiency of text analytics. The main idea of the technique is to enable direct document analytics on compressed data. This paper focuses on the programming challenges for developing efficient TADOC programs. It presents Zwift, the first programming framework for TADOC, which consists of a Domain Specific Language, a compiler and runtime, and a utility library. Experiments show that Zwift significantly improves programming productivity, while effectively unleashing the power of TADOC, producing code that reduces storage usage by 90.8% and execution time by 41.0% on six text analytics problems.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Information systems** → *Data analytics*;

KEYWORDS

Compilers, Domain Specific Languages, Text Analytics

1 INTRODUCTION

As data analytics becomes an increasingly important workload running on today's high-performance computing systems, how to improve the performance of high-performance data analytics is already an important topic in HPC [11, 21, 34]. Text analytics is an important class of high-performance data analytics problems, specializing on deriving statistics, insights, or knowledge from textual documents, such as system log files, textual content of web pages, phone or email records. It is essential for many domains, ranging from HPC system diagnosis, to health, security, and more.

This paper explores a recently-proposed approach to improving both the time and space efficiency of text analytics, called *text analytics directly on compressed data (TADOC)* [33]. The basic idea of TADOC is to conduct text analytics directly on compressed data, without recovering the original uncompressed data.

Even though data compression is commonly used in text analytics, all existing practices need to first decompress the data before processing them, which does not reduce but lengthen the processing time. An exception is Succinct [2]. Although Succinct processes compressed data directly, it is specific to the database domain, and it is designed mainly for the search and random access of arbitrary strings, rather than supporting general text analytics (more in Section 8).

By leveraging a compression algorithm called Sequitur [20], TADOC is feasible for general text analytics. Sequitur compresses a sequence of symbols into a Context-Free Grammar (CFG), which can be represented with a Directed Acyclic Graph (DAG). Text analytics can then be performed directly on the DAG. Compression by Sequitur takes time. Fortunately, many datasets (e.g., Wikipedia [1], UCI Machine Learning Repository [16]) are used for various analytics tasks by many users repeatedly. For them, compression time is well justified by the repeated usage of the compression results.

Directly working on compressed data, TADOC can save both time and space. Space-wise, its reduced data size saves not only the storage space, but also the amount of memory needed for processing the data. Time-wise, because it processes the compressed data where multiple duplications of an entry in the original dataset are already folded into one copy, TADOC can avoid repeatedly processing those duplications, and hence achieve speedup.

Unfortunately, developing programs to effectively apply TADOC to a given text analytics problem is challenging, especially for general programmers. The key to TADOC's time benefits is its reuse of processing results across multiple appearances of the same string in the original input. Such reuse helps avoid repeated processing. However, such reuse also requires the computation results to be saved and frequently propagated through the DAG. It is hence both important and tricky to strike a good tradeoff between the reuse and the overhead required to exploit it.

The problem is further complicated by the differences between the various analytics problems (e.g., some care about the boundaries of different files in the input dataset, some care about the order of words), the attributes of input datasets (e.g., sizes, numbers of files, unique words), and the demands for scalable performance. As a result, suitable ways (data structures, DAG traversal order, processing

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205325

algorithms) to implement TADOC differ across analytics problems as well as across input datasets. Although TADOC converts document analytics to graph problems, using general graph processing frameworks (e.g., Ligra [26]) does *not* solve these problems as these complexities are specific to text analytics; programmers would still need to deal with these complexities in their own code. A previous study [33] discusses the complexities and provides some guidelines for programmers to follow. However, developing efficient TADOC remains a difficult task for general programmers.

The objective of this work is to remove the major barriers to practical adoption of TADOC through the development of a programming system support. Specifically, we present Zwift, the first programming framework for TADOC. Zwift consists of an embedded domain-specific language (DSL), called the *Zwift Language*, a compiler and runtime, and a utility library.

Using the Zwift Language, programmers can easily express the *basic elements* of the text analytics of interest. These elements include data representation, domain of values, operators, direction, and are described in Section 5. The language also offers programmers the flexibility to write multiple (optional) ways to process the data if they are uncertain about which one works best. With the Zwift Language, programmers can concentrate on the functionality aspect of the data analytics solution without having to consider the many sophisticated performance factors.

The Zwift compiler automatically assembles the basic elements into an efficient program for TADOC. A major challenge is how to enable the produced code to employ the data structures and algorithms that best fit the specific analytics problems. Zwift addresses this challenge via a hybrid method that combines *rules* with *offline profiling*. Users have the option to provide multiple versions of code and some training inputs. The Zwift compiler generates an executable file for each version, measures the time on the training input, and selects the suitable version.

Zwift relieves programmers from most of the efficiency concerns and complexities for implementing TADOC. Our experiments on six text analytics problems demonstrate that Zwift significantly improves programming productivity: it reduces the number of lines of source code by 84.3%, and the program development time by 80%. At the same time, Zwift-produced code has performance that is comparable to or even better than that of manually-developed code. Overall, Zwift is effective in unleashing the power of TADOC: it reduces storage usage by 90.8% and execution time by 41.0%, compared to data analytics on uncompressed data.

2 BACKGROUND

This section provides background on TADOC, including the compression algorithm it builds on, and its basic idea.

2.1 Sequitur

The underlying compression algorithm of TADOC is Sequitur. Sequitur [20] is an algorithm for inferring 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. As it references all occurrences of the original string with the corresponding rule ID, the resulting CFG is usually more compact than the original input.

Figure 1 provides an illustration of Sequitur. Figure 1(a) shows the original input. Figure 1(b) shows the output of Sequitur in the form of a CFG. The CFG uncovers the repetitions in the input string as well as the hierarchical structure of the string. It uses R0 to represent the entire string, which consists of substrings represented by R1 and R2. The two instances of R2 in R0 reflect the repetition of "a b c a b d" in the input string. Similarly, the two instances of R1 in R2 reflect the repetition of "a b" in the substring of R2. 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. For efficiency, in Sequitur compression results, we represent each word with a unique non-negative integer, and 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 numeric representations of the words and rules in Figures 1(a, b), while Figure 1(e) shows the CFG in numerical form.

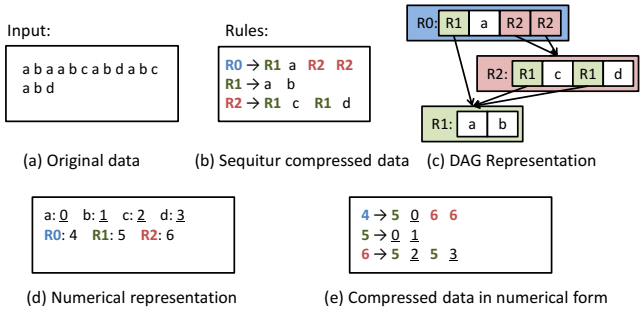

Figure 1: Illustration of Sequitur for compression.

Sequitur has several properties that make it appealing for our use. First, the CFG structure in its output 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., the distance used in LZ77 [36] and suffix array [19]). These properties make Sequitur a relatively easy fit for materializing the idea of compression-based document analytics. Note that we do not rule out the possible use of other compression algorithms for TADOC. The insights attained in this work hopefully could apply to other compression algorithms. A limitation of Sequitur is that its compression is relatively slow. However, as TADOC is designed for data with heavily repeated usage, compression time is not a major concern.

2.2 Basic Idea of TADOC

We take *word count* as an example to explain the basic idea of TADOC.

Word count [3, 7, 22] is a basic algorithm widely used in document classification, clustering, and theme identification. It counts the total appearances of every word in a given dataset which often consists of a number of files. Its inputs and outputs are as follows:

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

Figure 2 shows a possible implementation for *word count* with TADOC. We use the DAG that Sequitur generates on the string shown in Figure 1 (a) as the example. The processing consists of two main steps. The first step calculates the frequency with which each rule appears in the entire dataset (① to ③ in Figure 2). This step is performed via a preorder traversal of the graph: parent nodes pass their frequencies to their children, from which, the child nodes can calculate their frequencies. With this step done, the second step (④ in Figure 2) just needs to go through each of the rules once (with no need for another graph traversal). When it goes through a rule, it can use the total frequencies of the rule to calculate the local frequencies of a word. Here, the local frequency of a word is the directly-observable frequency of the word in the right-hand side of the rule (i.e., local word table, without considering its subrules), multiplied with the frequency of that rule (weight). By summing up the local frequencies across all rules for each word, this step provides the final results. For example, the local frequency of word a is 1 in R0 (weight: 1, local word table: <a,1>, 1×1), and 5 in R1 (weight: 5, local word table: <a,1>, 5×1), so the word a's total frequency is 6. Similarly, the word b has a total frequency of 5.

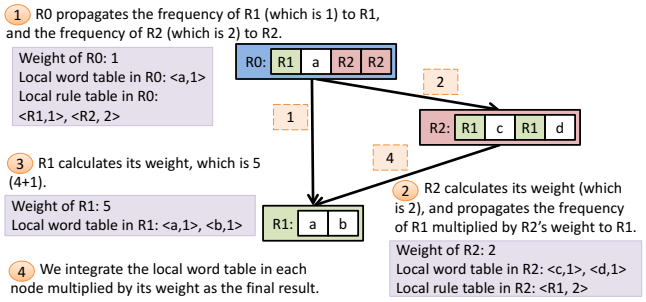

Figure 2: Illustration of a preorder traversal algorithm for *word count* on the same string as Figure 1 deals with. The weights on an edge between two nodes (say, $node_x$ to $node_y$) indicate the total appearances of $node_y$'s rule incurred due to all appearances of $node_x$'s rule.

Note that there are other possible implementations. For example, we can traverse the graph in postorder (children before parents): Each node counts local word frequency (i.e., the word counts immediately contained in this node's rule without considering subrules) and then adds the word counts from its children nodes to its own word counts. The word counts obtained in the root node give the final results.

3 PROGRAMMING CHALLENGES

Developing an efficient program for effectively applying TADOC to a given text analytics problem is challenging, especially for general programmers. It fundamentally faces a cost-benefit tradeoff that is determined by many factors, ranging from what data structures to use for saving and propagating information across the DAG, to the traversal order of the DAG, the different properties and requirements of different text analytics problems, the attributes of the input datasets, and so on. The seminal study of TADOC [33] describes complexities in three main dimensions, which we list as follows.

(1) *Problem dimension.* Different analytics problems have different requirements and complexities. We use two other analytics problems to explain.

Inverted index [3, 9, 29] builds word-to-file indices 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. So unlike *word count*, which treats inputs as a bag of words, *inverted index* distinguishes one file from another.

Sequence count [3, 15, 31] is another example problem. It counts the number of appearances of every l-word (e.g., l=3) sequence in each file. *Sequence count* is very useful in semantic, expression, and sequence analysis. Compared to *word count* and *inverted index*, *sequence count* needs to not only distinguish between different files, but also discern the order of consecutive words.

Due to these special requirements, the algorithm in Figure 1 cannot work for these two problems. The programmer must redesign the algorithm to somehow identify and carefully handle the file boundaries. For *sequence count*, the algorithm must, in addition, be aware of the order of appearance of words in the DAG. As an l-word sequence may span the boundaries of multiple nodes in the DAG, the algorithm must treat the boundaries and order of nodes with caution.

(2) *Implementation dimension.* As shown in the *word count* example in Section 2.2, there are multiple ways to implement TADOC for a given data analytics problem. These different implementations may use different orders for graph traversal, and employ different data structures to store and pass intermediate data across the DAG. As a result, they may save different amounts of processing time and also incur different amounts of overhead. The best decision may be hard to determine as it may depend on the attributes of the input datasets (as discussed next). In addition, there are different types of execution platforms, ranging from parallel to distributed environments. Writing parallel or distributed code is often difficult for general programmers; requiring a version for each type of platform adds more difficulty for both code development and maintenance.

(3) *Dataset dimension.* The attributes of input datasets may sometimes substantially affect the overhead and benefits of a TADOC implementation. For instance, when solving *inverted index*, one method is to propagate the list of files in which a word appears through the graph. This approach could work efficiently if there are a modest number of files, but would incur substantial propagation overhead when the number of files is large as the list to propagate could get very large. So, datasets of different properties could demand a different design in what to propagate and the overall traversal algorithm.

These complexities make it challenging for general programmers to adopt TADOC. Our experience shows that a poorly written TADOC program could easily cause the overheads that outweigh the benefits. These difficulties motivate our development of *Zwift*.

4 A CONCEPTUAL FRAMEWORK

To create a programming framework for TADOC, we need to first get a unified view of the various TADOC problems. Doing so could allow us to map the many different problems to the same abstraction, and hence offer a unified programming framework.

The view we get is inspired by the classic *data flow framework* in compilers, which unifies the treatment of various data flow problems. Through examination of various text analytics problems, we create *a unified conceptual framework for TADOC* (*unified framework* in short). It captures the common operations and workflows of TADOC through a six-element tuple (G, V, F, $\bigvee$, D, $\bigwedge$), defined as follows:

(1) A graph G, which is the DAG representation of the compression results of Sequitur on a dataset.
(2) A domain of values V, which defines the possible values as the outputs of the document analytics of interest.
(3) A domain of values F, which defines the possible values to be propagated across G.
(4) A meet operator $\bigvee$, which defines how to transform values in F at one step of the traversal of G.
(5) A direction of the value flow D, which is FORWARD (parents before children) or BACKWARD (children after parents) or DEPTHFIRST (in the order of the original files).
(6) A gleaning operator $\bigwedge$, which defines how to transform values in V in the final stage of the analytics.

The solutions to the various document analytics problems listed in the previous sections can all be mapped to this abstract model. Each can be regarded as an instance of the *unified framework*, with the six components instantiated using values specific to the analytics problem.

Take *word count* as an example. For that problem, G is the DAG of the Sequitur compression results on the dataset of interest, and V is the set of $< w, n >$, where w is a possible word, and n is a non-negative integer. The two algorithms described in Section 2.2 for solving the *word count* problem are essentially different instances of the *unified framework* with different definitions of F, D, $\bigvee$, and $\bigwedge$. The solutions to the other two analytics problems outlined in Section 3 can be mapped to the *unified framework* in a similar manner. For sequence counting, for instance, G is the DAG of Sequitur results, V is $< s, n >$ (s represents a l-long word sequence, n a non-negative integer), F contains empty operation, D can be DEPTHFIRST, $\bigvee$ increases counts in some global or local tables to record intermediate results, $\bigwedge$ calculates frequencies of rules and combines recorded results when necessary.

High-Level Algorithms for Solving Sequitur-Based Document Analytics Problems. After abstracting various concrete analytics problems into our *unified framework*, we can examine the solutions to various Sequitur-based document analytics problems in a single view. Despite their differences, all the solutions to these problems are essentially variants of the following high-level algorithm.

`High-level algorithm for Sequitur-based document analytics:`

(1) *Loading:* Load Sequitur grammar, build G (or its variants), and initialize the data structures local to each node or global to the algorithm.
(2) *Propagation:* Propagate information with the meet operator $\bigvee$ while traversing G in direction D.
(3) *Gleaning:* Glean information through the gleaning operator $\bigwedge$ and output the final results.

We find that this high-level algorithm suits all document analytics problems we have encountered. The main differences among the different solutions to a problem are mostly about the information they propagate, the meet and glean operators they use, the direction of the traversal, and the assisting data structures they employ. These aspects together cover the set of most important factors that determine the efficiency of the constructed solutions, and lay the basis of our design of the Zwift programming framework.

5 THE ZWIFT LANGUAGE

Based on the conceptual framework in the previous section, we develop a DSL, called *Zwift*, for programming text analytics to make use of TADOC with ease. To ease understanding, we explain the Zwift DSL through informal intuitive descriptions and examples rather than formal language semantic definitions. Listing 1 shows the main constructs in Zwift.

Listing 1: Zwift DSL template.

```
ELEMENT = LETTER/WORD/SENTENCE
USING_FILE = true/false
NodeStructure = {
//data structures in each node
}
Init = {
//initializations for each node in ZwiftDAG
}
Direction = FORWARD/BACKWARD/DEPTHFIRST
Action = {
//actions taken at each node during a traversal
}
Result = {
// result structure
}
FinalStage = {
// final operations to produce the results
}
```

These constructs correspond to the elements in the *unified framework* presented in the previous section.

(1) `ZwiftDAG` represents the DAG from Sequitur, corresponding to G in the *unified framework*.
(2) `Result` represents the structure of the analytics result, which corresponds to V in the *unified framework*.
(3) `NodeStructure` represents the user-defined data structures in each node (F in the *unified framework*). The construct `Init` is for programmers to define the initialization of the data structures in each node.
(4) `Action` represents the operations to take in each node during the traversal of the DAG ($\bigvee$ in the *unified framework*).
(5) `Direction` represents the direction of data propagation (D in the *unified framework*).
(6) `FinalStage` defines how to get the final result (the gleaning operator $\bigwedge$ in the *unified framework*). Zwift has an optional section, FinalMerging, which is about how to merge the results from FinalStage when the inputs are partitioned into multiple chunks.

Currently, Zwift is implemented as a DSL embedded in C/C++. It allows only C/C++ code inside the Zwift code constructs (e.g., `Init`, `Action`, `FinalStage`). Adding support of Python or other programming languages is our future work.

In addition, Zwift has some reserved keywords, shown in Table 1. These keywords are in four categories. The first category is for *indicators*. Users set these keywords to indicate certain attributes

Table 1: Some reserved keywords in Zwift.

Type	KeyWord	Description
indicators	ELEMENT	granularity of input: LETTER/WORD/SENTENCE. ELEMENT is also a data type
	USING_FILE	whether the DAG needs to include file information; default: false
	ZwiftNodesExceptRoot	if true, the root node shall *not* perform the code in `Action`; default: false
	NODE_COARSENING	node coarsening threshold; default: 0 for no coarsening
	EDGE_COARSENING	whether Zwift performs edge coarsening; default: false
	INPUT_PATH	the directory path of the compressed data; default: the first command-line parameter
data type	RULEID	data type of rule ID
	FILEID	data type of file ID
	ZwiftMap	Zwift map data structure
	ZwiftVec	Zwift vector data structure
	ZwiftSet	Zwift set data structure
replacement	ROOT	the root rule ID
	NODE	the node itself; this will be replaced by the compiler
	CHILD	child node; this keyword will be replaced by the compiler
	PARENT	parent node; this will be replaced by the compiler
built-in variables	ZwiftDAG	the DAG representation of the compression results
	FILES	number of files
	RULES	number of rules
	WORDS	number of words
	ruleID	the ID of the rule, use NODE.ruleID to get it

of the analytics problem (e.g., whether the analytics discerns file boundaries). The second category is *data types*. Some of these are abstract data structures, which get instantiated by the compiler and runtime based on the attributes of the analytics problem and the uncovered properties of the input datasets. An example is `ZwiftVec`, which could be substantiated with a regular C++ vector, a one-level bit vector, or a two-level bit vector as Section 6.2 elaborates. The third category is for the keywords for *replacement*. The Zwift compiler replaces these keywords during compilation. The fourth category consists of some *built-in variables*, such as the number of files, rules, and so on. User code can use these structures without defining them; the compiler inserts code to define their values at runtime.

We take *word count* as an example and show its Zwift code in Listing 2. Our code first indicates that the application works at the word level and does not need to discern file boundaries. It then gives the compiler the permission to optimize the DAG by coarsening its edges (explained later in this section). It defines `NodeStructure` as a structure consisting of a `wordCount` table, a `ruleCount` table, and an integer field named "weight". Note that during compilation, the Zwift compiler automatically adds some extra fields into `NodeStructure`, including an integer field `ruleID` corresponding to the rule ID of the node, and several other fields to assist the runtime traversal order of the nodes in the DAG, as the next section will explain. `NodeStructure` is initialized through the `Init` block, where the keyword `NODE` indicates the node itself and will be replaced by the compiler with actual variables, such as "ZwiftDAG.Nodes[i_zwift]". `Zwift_IsWord()` checks whether an element in a rule is a word or a rule. The code sets the DAG traversal direction to `FORWARD`. Zwift has three predefined traversal directions; `FORWARD` means a preorder (parents before children), `BACKWARD` means a postorder (children before parents), `DEPTHFIRST` means a depth-first traversal of the DAG. The `Action` block specifies the actions to take for each node during the DAG traversal. The construct `Result` defines the structure of the final output. The construct `FinalStage` specifies the actions to take at the end of the DAG traversal.

Listing 2: Word Count using Zwift.

```
1  ELEMENT = WORD
2  USING_FILE = false
3  EDGE_COARSENING = true
4  NodeStructure = {
5    ZwiftMap<ELEMENT, int> wordCount;
6    ZwiftMap<RULEID, int> ruleCount;
7    int weight;
8  }
9  Init = {
10   if (NODE.ruleID == ROOT) {
11     NODE.weight = 1;
12   }else{
13     NODE.weight = 0
14   }
15   for(vector<int>::iterator i=NODE.input.begin();
16       i!=NODE.input.end();
17       i++) {
18     if( Zwift_IsWord(*i) )
19       NODE.wordCount[*i]++;
20     else
21       NODE.ruleCount[*i]++;
22   }
23 }
24 Direction = FORWARD
25 Action = {
26   CHILD.weight +=
27     PARENT.weight * PARENT.ruleCount[CHILD.ruleID];
28 }
29 Result = {
30   ZwiftMap<ELEMENT, int> result;
31 }
32 FinalStage = {
33   for(int i=0; i<RULES; i++){
34     for(ZwiftMap<ELEMENT, int>::iterator
35         j = ZwiftDAG.Nodes[i].wordCount.begin();
36         j != ZwiftDAG.Nodes[i].wordCount.end();
37         j++)
38       result[j->first]+=(j->second)*ZwiftDAG.Nodes[i].weight;
39   }
40 }
```

Zwift has an optional section of `GlobalVariables` for programmers to define some global variables, and a related section of `GlobalVariableInit` for initializing those variables.

When a programmer does *not* know the optimal implementation (the optimal may be input dependent), he/she can provide several versions of Zwift code for constructs `Init`, `Direction`, `Action`, and `FinalStage`. The version numbers are expressed as subscripts of the constructs (e.g., `Action[0]` and `Action[1]`). The next section explains how the Zwift framework selects the appropriate version on the fly.

6 ZWIFT COMPILER AND RUNTIME

The Zwift compiler and runtime play a critical role in creating efficient sequential, parallel, or distributed system code for applying TADOC based on the Zwift code. This section first gives an overview of the compiler and explains its basic functionality of assembling the Zwift code. It then elaborates on the important optimizations conducted by the compiler and runtime through data abstraction, coarse-grained partitioning, and version selection.

6.1 Overall Structure and Basic Functionality

The Zwift compiler takes in code written in the Zwift Language and generates the corresponding C/C++ code. It uses the code skeletons we have constructed based on the common patterns seen in TADOC programs. Listing 3 outlines the simplified code skeleton for the sequential version. There are also code skeletons for parallel and distributed code in Pthreads and Spark, respectively. At a high level, the compilation goes through the following four main steps: (1) The compiler reads in the Zwift code and performs static analysis; it replaces some of the keywords (e.g., ROOT, NODE), instantiates the abstract data structures (e.g., ZwiftVec, ZwiftMap), and completes the data structures and code blocks in the Zwift code (e.g., by adding some counters into `NodeStructure` for dependence tracking, as explained later); (2) The compiler inserts the completed code blocks into the code skeleton that conforms to the traversal direction specified in the Zwift code; (3) The compiler converts the code to parallel and distributed versions by leveraging the corresponding code skeletons. When there are multiple choices of the `Action` code blocks, the compiler generates multiple versions of the program with each version corresponding to one of the choices. (4) The programmers can do profiling runs on the multiple versions and create a version selector for runtime adaptation.

Some of the operations are worth further explanation, in particular: the treatment of different traversal orders in the code generation process, the instantiation of the abstract data structures, version selection, and the creation of the parallel and distributed versions. We explain the *traversal order* construction first and then use separate subsections to elaborate on the other aspects.

Zwift currently has three predefined **traversal directions** of DAG, forward, backward, and depth-first. The compiler generates respective code to materialize each of them.

Forward traversal requires a preorder (parents before children). The Zwift compiler inserts extra fields into `NodeStructure` and employs a ready queue to efficiently ensure the correct visiting order of nodes. Specifically, it adds a field `INEDGES` and a field `CNT` into `NodeStructure`. At graph loading time, the `INEDGES` of each

Listing 3: Sequential code skeleton used by Zwift compiler.

```
#include"Zwift.h"
...
struct NodeStructure{
//  <- insert the NodeStructure in Zwift code here
  ...
}
void init(){
  ...
  for(vector<int>::iterator i_zwift = ZwiftNodes.begin();
    i_zwift != ZwiftNodes.end();
    i_zwift++) {
//  <- insert the Init block of Zwift code here
  }
  ...
}
// Zwift provides different functions for action,
// which relates to direction, such as
// ForwardAction(), BackwardAction(). We take
// ForwardAction() as an example
void ForwardAction(){
  ...
// construct for-loop, using Listing 2 as example
  for(ZwiftMap<int, int>:: iterator i_zwift
      = ZwiftNodes[head].ruleCount.begin();
    i_zwift=ZwiftNodes[head].ruleCount.end();
    i_zwift++){
//  <- insert the action in the Zwift code here
  ...
  }
  ...
}
void FinalStage(){
//  <- insert the FinalStage in Zwift code here
}
int main() {
  ...
  init();
  ForwardAction();
  FinalStage();
  ...
}
```

node is set to the total number of incoming edges (i.e., parents) of the node. During graph traversal, the `CNT` field of a node tracks the number of nodes' parents that have been processed. Once `CNT` equals `INEDGES`, this node is put into the ready queue. Only nodes in the ready queue can get processed. Initially, only the root node is in the queue. A node is removed from the queue after it is processed. When the queue is empty, the traversal is done.

In the Zwift DAG, a parent node knows which nodes are its children, but a child node has no knowledge about its parents. Backward traversal is, therefore, implemented with recursive calls. The calls start from the root node and recursively reach the leaves where the actual processing gets started. Because a node can be the child of multiple parent nodes, the Zwift compiler inserts code to track the status of a node at runtime to avoid processing a node repeatedly.

Depth-first traversal is particularly useful for analytics (e.g., *sequence count*) that are sensitive to the original appearance order of words in the input. It ensures that the traversal order is in the same order of appearance of the words in the original input, to facilitate the treatment of sequences spanning node boundaries. Depth-first traversal is implemented by recursively processing each item in each rule. Reuse of intermediate results across nodes is still possible. For example, in *sequence count*, the frequency of the word sequences within a node are stored in a local data structure for later reuse.

As Zwift takes care of all such implementation details, it allows programmers to focus only on the primary functionality aspects

and the goals of the analytics problem and solution. We next describe some key optimizations performed by the Zwift compiler and runtime for a high computational efficiency.

6.2 Data Abstractions

One feature of Zwift is its use of abstract data structures for compile-time and run-time optimizations. Programmers use these abstract data structures in their Zwift code, without worrying about their efficiency. The compiler instantiates them with one of several optional concrete data structure implementations during code transformations. It makes the selection based on the attributes of the analytics problem. It inserts code such that, at runtime, based on the properties of the input datasets, the code automatically configures the use of these concrete data structure implementations to maximize performance. We explain the optimization through two of the abstract data structures, `ZwiftVec` and `ZwiftDAG`.

ZwiftVec. A `ZwiftVec` can be instantiated as a regular vector, a one-level bit vector (regular bit vector), or a two-level bit vector. The compiler makes the decision based on the problem attributes specified in the given Zwift code for high efficiency. It uses the bit vectors when a large set is needed to record Boolean information. For example, for file-sensitive problems such as *inverted indexing*, the algorithm needs to propagate across the DAG the set of files that contain each word. One possible design is that each node uses a *set* structure to store the IDs of the files containing a certain word. Frequent queries and insertions to the set could cause large overheads to the processing. An alternative design is that each node uses a bit vector, with each bit corresponding to a file: 1 for file presence, 0 otherwise. Such representation can help replace the slow set operations with fast bit operations. However, it does not work for all cases. When the number of files is very large, this design could incur large space overhead, as every node needs such a vector, and the length of each of the vectors needs to be the same as the total number of files.

Zwift addresses the problem by adopting a two-level bit vector, illustrated in Figure 3. Level Two contains a number of N-bit vectors (where N is a configurable parameter). 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 few files, then most elements in its pointer array would be *null*. For the level-1 bit vector, only the elements corresponding to that rule have starting address. For level-2 bit vectors, only the bits associated with files have values. The two-level bit vector requires more indirect references than a one-level vector does. But, it uses much less space. The number of elements in the first-level arrays and vectors of the two-level bitmap is only $1/N$ of the number of files.

Zwift includes all three kinds of implementations for `ZwiftVec`. At compile time, it decides whether a regular vector is going to be used based on whether the vector is used as a set for Boolean information. If that is the case, it postpones the decision of using one-level or two-level bit vector to the Zwift runtime. It does that by creating multiple versions of the code, each using a different bit vector implementation for a `ZwiftVec`. At runtime, after the graph

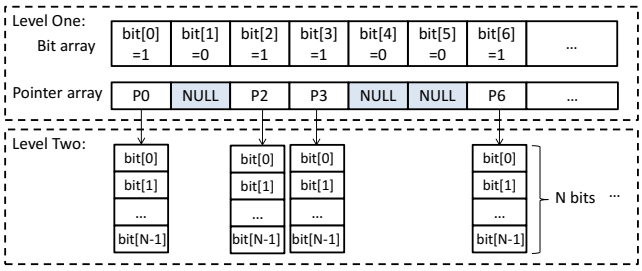

Figure 3: An illustration of the two-level bit vector for both memory footprint minimization and access efficiency.

is loaded, the runtime calculates the space cost of using a one-level bit vector based on the graph size. If the cost is over a threshold (e.g., half of the available memory), it selects the two-level bit vector version of the code.

ZwiftDAG. `ZwiftDAG`, the abstract data structure for the loaded DAG, features an optimization, *coarsening*, which includes edge merging and node coarsening.

As an abstract data structure that programmers use in their Zwift code, `ZwiftDAG` is a data structure to hold the loaded DAG. It gets materialized at compile time. The structure of its node is determined by `NodeStructure` in Zwift code (after the compiler adds extra fields). User code can use `ZwiftDAG.Nodes` and `ZwiftDAG.Edges` to represent all the nodes and edges in the DAG. For example, Listing 2 uses `ZwiftDAG.Nodes` to summarize the word counts in `FinalStage`.

Coarsening is an optimization that happens during the graph loading time. Through it, the nodes or edges in the Zwift DAG can represent the aggregated information of a set of nodes or edges. The two coarsening methods, *edge merging* and *node coarsening*, can be used together by Zwift runtime. *Edge merging* merges multiple edges between two nodes into one, and the weight of the edge may be used to indicate the number of original edges in the Zwift code. Edge merging loses the order of words, but helps reduce the size of the graph and hence the number of memory accesses during graph traversal. It is helpful for analytics tasks that are insensitive to word order (e.g., *word count* and *inverted index* but not *sequence count*). *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. *Node coarsening* 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 reduce memory space usage and graph traversal time.

The Zwift Language provides constructs, `NODE_COARSENING` and `EDGE_COARSENING`, to allow users to indicate their preferences for coarsening, as shown in Table 1. Our compiler automatically generates code to load user's data files to instantiate ZwiftDAG, and does the coarsening at loading time based on user's specifications. To enable edge coarsening, the user sets `EDGE_COARSENING` to `true`; to enable node coarsening, the user sets `NODE_COARSENING` to a

positive number, which serves as the threshold of the minimum size of a node (i.e., the total size of all its elements). For nodes below that threshold, their contents are folded into their parent nodes at the graph loading process, starting from leaf nodes.

6.3 Parallel and Distributed Code Generation

To obtain scalable performance, Zwift generates parallel and distributed code versions to leverage parallel and distributed computing resources. We have explored two methods for parallelism via Zwift. The first method is a fine-grained method to distribute rules to different threads, which mainly parallelizes the `Action` part in the Zwift code. For instance, for forward processing, the compiler could generate code to allow multiple threads to simultaneously process the nodes in the ready queue. This method does not give us significant parallel performance, due to the synchronization overhead and the limited amount of parallelism. Some sophisticated parallel graph processing techniques [33] may potentially help; we leave in-depth exploration to future work.

The second method explores coarse-grained parallelism, which provides much better results. The parallelism is at the data level. When users use the Zwift utility library (described in Section 6.5) to compress data, by setting some parameters of the call, they can let the compression API to first partition the input data into smaller partitions of similar sizes, which can then be processed by different threads or processes in parallel. To take advantage of this feature, programmers need to add one extra construct, `FinalMerging`, into the Zwift program. This construct specifies how to merge the processing results of the partitions into the final output.

The coarse-grained method may lose some opportunities for reusing results as reuse now happens only within each partition. However, it avoids the frequent synchronization the fine-grained method suffers from. Because typical text analytics problems use large datasets, our experiments show that the coarse-grained method usually provides better and more scalable performance. On *word count*, for instance, the fine-grained parallel version runs 50% slower than the coarse-grained version even on a small dataset (dataset A in Table 4).

The coarse-grained method also simplifies the conversion from sequential code to parallel and distributed versions of the code. Because there is no communication needed between the processing of different partitions, the compiler only needs to let each thread or process run the same sequential code on one partition, and call the merging block to conduct the final merging. Our implementation uses a skeleton in Pthreads for the parallel version and a skeleton in Spark for the distributed version. For the Spark version, Zwift uses Spark pipe() [30] to directly call the sequential version.

We note that a side benefit of the coarse-grained method is that it also allows parallel compression of the partitions of the input data, which helps shorten the compression process.

6.4 Version Selection

As explained in Section 5, for a given application, programmers may provide multiple Zwift `ACTION` code versions with different DAG traversal orders and implementations for a text analytics problem. Based on the source code, it can be hard to choose the best implementation, because the performance of many applications depends on the input dataset.

The solution currently employed in Zwift is a hybrid method that combines rules with offline profiling. The primary rule is that if the problem is an order-sensitive problem, then `DEPTHFIRST` should be the only choice, because `BACKWARD` and `FORWARD` do *not* traverse the DAG in the original file order. This rule comes from our earlier discussions on the special needs of such problems. Other cases are addressed currently through a synergy between Zwift and programmers. By offering the support for automatically generating multiple versions of the program (as aforementioned), Zwift allows programmers to easily run the different versions on some representative inputs. After observing the performance of the different versions, the programmer may either settle on one of the versions or write a wrapper to check certain conditions of the given input dataset to select the appropriate version to run.

A potential improvement is to automate the version selection process by deriving an analytical performance model for various versions of `ACTION` on DAGs, and then, at runtime determining the best version to use by plugging into the analytical performance model the actual features of the current input datasets. We leave a thorough exploration of this direction to future work.

6.5 Extra Optimization

An extra optimization we find helpful is to perform further compression (e.g., using Gzip) of the compressed results from Sequitur. Doing so adds some time overhead for data processing as the Sequitur results need to be first restored before TADOC starts. However, we find that the time overhead is orders of magnitude smaller than restoring the original data because Sequitur results are much more compact than the original data. This optimization significantly saves storage space, as Section 7 shows.

7 EVALUATION

In this section, we use six benchmarks and five datasets to evaluate the performance of TADOC and the effectiveness of Zwift in supporting TADOC, in terms of both programming productivity and code performance.

7.1 Benchmarks

We use text analytics benchmarks from two publicly-available benchmark suites, HiBench [13] and Puma [3].

Word Count. This program counts word frequencies, as Section 2.2 already describes. Its Zwift code is shown in Listing 2 and discussed in Section 5.

Inverted Index. This program identifies, for each word, the set of files containing it, as Section 3 describes. We show its Zwift code in Listing 4. It contains two implementations, `Action[0]` and `Action[1]`, and there are two `FinalStages` related to these two different traversals. The Zwift code, equipped with a version selection module created through offline profiling, automatically selects one of the versions to use for a given input dataset.

Sequence Count. This program counts, for each file, the frequencies of every distinct three-word sequence it contains. It is explained in Section 3. The graph traversal order is DEPTHFIRST.

Ranked Inverted Index. This program produces a list of three-word sequences in decreasing order of their occurrences in each document. Similar to *sequence count*, it is also order-sensitive. Its direction is hence set to DEPTHFIRST. Its FinalStage contains a step to rank all sequences based on their counts.

Listing 4: Inverted Index using Zwift.

```
ELEMENT = WORD
USING_FILE = true
EDGE_COARSENING = true
NodeStructure = {
  ZwiftSet<ELEMENT> wordSet;
  ZwiftSet <RULEID> ruleSet;
  ZwiftVec <FILEID,ZwiftBit> file;
}
Init = {
  if(RULEID ==ROOT) {
    for(int i=0; i<FILES; i++){
      int start=getPartition(i);
      int end=getPartition(i+1);
      for(int j=start; j<end; j++){
       if(Zwift_IsWord( ZwiftDAG.Nodes[0].input[j]))
        result[ZwiftDAG.Nodes[0].input[j]].insert(i);
       else
        ZwiftDAG.Nodes[j].file[i]=1;
      }
    }
  }
  else {
    for(vector<int>::iterator i=NODE.input.begin();
       i!=NODE.input.end(); i++) {
      if( Zwift_IsWord(*i) )
        NODE.wordSet.insert(*i);
      else
        NODE.ruleSet.insert(*i);
    }
  }
}
ZwiftNodesExceptRoot = true
Direction[0] = FORWARD
Direction[1] = BACKWARD
Action[0] = {
  for(ZwiftVec <FILEID,ZwiftBit>::iterator i=
      PARENT.file.begin();
      i!= PARENT.file.end(); i++){
    CHILD.file[*i]=1;
  }
}
Action[1] = {
  for(ZwiftSet <ELEMENT>::iterator i=
       CHILD.wordSet.begin();
     i!= CHILD.wordSet.end(); i++){
    PARENT.wordSet.insert(*i);
  }
}
Result = {
  ZwiftMap<ELEMENT, ZwiftSet<FILEID> > result;
}
FinalStage[0] = {
  for(int i=1; i<RULES; i++){
    for(ZwiftSet<ELEMENT>::iterator j=
          ZwiftDAG.Nodes[i].wordSet.begin();
        j != ZwiftDAG.Nodes[i].wordSet.end(); j++){
      for(ZwiftVec <FILEID,ZwiftBit>::iterator k=
            ZwiftDAG.Nodes[i].file.begin();
          k != ZwiftDAG.Nodes[i].file.end(); k++)
        if(ZwiftDAG.Nodes[i].file[*k]==1)
          result[*j].insert(*k);
    }
  }
}
FinalStage[1] = {
  for(int i=0; i<FILES; i++){
    int start=getPartition(i);
    int end=getPartition(i+1);
    for(int j=start; j<end; j++){
      if(!Zwift_IsWord(ZwiftDAG.Nodes[0].input[j]))
        for(ZwiftSet <RULEID>::iterator k=
            ZwiftDAG.Nodes[j].wordSet.begin();
            k != ZwiftDAG.Nodes[j].wordSet.end();
            k++)
          result[*k].insert(i);
    }
  }
}
```

Sort. This program sorts words in alphabetical order. In Zwift code, the words are stored in the abstract data structure ZwiftVec. Its FinalStage calls a sorting function on the vector.

Term Vector. This program outputs the most frequent words in each file. It needs to keep the file information, so one strategy is to make ZwiftDAG transmit the file information from the root to the other nodes first, and then perform *word count* on the files. However, this strategy needs to maintain a copy of the word count of each file in each node, which is costly when there are many files. An alternative strategy is to transmit word counts from children to parents in a backward order but not include the root node by using ZwiftNodesExceptRoot. After all nodes finish the "Action", the word frequencies of other nodes (all nodes except the root) are stored in the root's direct children. Because the root node contains file information, using the root node and its direct children can easily calculate the word count of each file and then generate the most frequent words. The code for this version shares some similarity with the backward version of *inverted index*. It can contain both a backward and a forward version.

7.2 Programming Productivity

Table 2 shows the number of lines of code needed in C++ and in Zwift. The "C++" column shows the length of the code written for TADOC without using the Zwift framework; it employs existing libraries (e.g., the C++11 Library) that may help shorten the code. The "Zwift DSL" column shows the length of the equivalent Zwift code. On average, the Zwift code is 84.3% shorter than the equivalent C++ code. Such a comparison is based only on the sequential code. If we count the parallel and distributed versions, the savings are even greater, because the sequential version of Zwift code is sufficient for the Zwift compiler to generate the parallel and distributed versions, while without Zwift, users would need to write *all three versions* separately. In addition, as the Zwift programmer does not need to worry about the graph traversal implementation and the various optimizations related to efficiency, our experience is that the average time per line of code for Zwift is shorter than that for the C++ code. Therefore, we expect that Zwift's time savings for code development would be even greater than the reduction in lines of code.

Table 2: Lines of code in C++ vs. Zwift.

Programs	C++	Zwift DSL
Word Count	267	70
Sort	279	84
Inverted Index	616	78
Term Vector	719	87
Sequence Count	451	28
Ranked Inverted Index	440	31

7.3 Performance

We measure the performance of Zwift on two platforms. One is a single node machine, and the other is a ten-node cluster on Amazon EC2 [4], as shown in Table 3. We test small datasets on the single-node machine with the sequential version of code produced by Zwift. For large datasets, we use the ten-node Spark cluster with

the HDFS storage system [8], and run the distributed code produced by Zwift.

Table 3: Platform configurations.

Platform	Single Node	Spark Cluster
Operating System	Ubuntu 14.04.2	Ubuntu 16.04.1
Compiler	GCC 4.8.2	GCC 5.4.0
Number of Nodes	1	10
CPU	Intel i7-4790	Intel E5-2676v3
Cores/Machine	4	2
Frequency	3.60GHz	2.40GHz
MemorySize/Machine	16GB	8GB

Table 4 describes the datasets we use in our experiments. Dataset A is NSF Research Award Abstracts (NSFRAA) from the UCI Machine Learning Repository [16]. This dataset consists of a large number of small files (134,631), and we use this dataset to measure Zwift's ability to utilize duplicated information among small files. Dataset B to E are collections of web documents from the Wikipedia database [1]. These datasets consist of large files.

Table 4: Tested datasets. Size is for the original datasets.

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

We measure the space and time benefits in the following two sections. We test the performance of datasets A and B on a single node and datasets C, D and E on the cluster. We have implemented four versions: (1) *manual-direct*, which is the baseline text analytics technique on *uncompressed* data; (2) *manual-gzip*, which is the baseline text analytics technique that uses Gzip to compress data for space savings and decompress the compressed data during processing time; (3) *manual-opt*, which is the version we *manually* developed to apply TADOC on our Sequitur-compressed data with all the optimizations we described in the paper except version selection; in case of multiple possible directions to implement, *manual-opt* just uses the one that works best for most of our testing datasets; (4) *Zwift*, which is the code using our DSL with all the optimizations, including version selection. We use *manual-direct* as the common baseline for comparison.

7.3.1 Storage Benefits. Table 5 reports the storage savings of *manual-gzip* and Zwift. Compared to the original uncompressed datasets, Zwift reduces storage usage by 90.8%, even more than *manual-gzip* does. This is because the Sequitur-based algorithm provides the first round compression, which makes the data smaller than the original size, and then the second-round compression by Gzip makes the data even more compact. Table 5 also shows that larger datasets can be compressed better with Zwift.

Table 5: Storage savings.

Dataset	manual-gzip (%)	Zwift (%)
A	82.93	84.67
B	88.75	91.61
C	89.20	92.90
D	88.71	92.47
E	88.23	92.38

7.3.2 Time Benefits. We use speedup over $Time_{original}$ (i.e., execution time of *manual-direct*) as the metric to quantify time benefits. The *manual-gzip* version suffers from decompression time. Even if decompression is done by a separate thread that runs in parallel with the processing threads, we find that *manual-gzip cannot* run faster than *manual-direct*, since it adds extra operations while saving none. Therefore, we focus our discussion on the comparisons between other versions.

We show the time benefits of *manual-opt* and Zwift in Figure 4. Zwift yields 2X speedup, on average, over *manual-direct*, thanks to its reuse of processing results throughout the DAG. Some applications on large datasets, such as *term vector* on dataset C, show more than 70% performance benefit. The reason is that the resilient distributed dataset (RDD), which is the basic storage unit in Spark[1], has a size limit. Some of the original files exceed the RDD limit and have to be split into separate RDDs. For programs that need file information, such as *inverted index* and *term vector*, additional data structures are needed to be created and maintained to record file information, which adds significant execution time cost. Zwift compression decreases the file sizes, enabling most of these large files to fit into the size of RDD after compression, and hence avoids the large execution time cost.

From Figure 4, we can see that the time benefits of Zwift are nearly the same as what *manual-opt* gives in almost all cases, which indicates that Zwift successfully unleashes most of the power of TADOC, while avoiding the manual programming and optimization burden. The only exceptions are dataset B on *inverted index* and *term vector*, in which case, the benefits from Zwift are *even larger* than what *manual-opt* provides. There two reasons for Zwift's larger benefits. First, Zwift successfully applies *all* the optimizations that *manual-opt* does *without* adding noticeable extra overhead. Second, version selection by Zwift exhibits additional performance benefit, making Zwift outperform *manual-opt*. Recall that *manual-opt* does not perform version selection. Instead, it implements backward traversal for *inverted index* and *term vector*, as backward traversal works the best on them on most datasets. Dataset B is an exception as it contains only four large files. The forward version has a lower cost for both benchmarks. Zwift's version selection mechanism helps it select the right version to run, improving performance.

The offline profiling of version selection in Zwift costs negligible overhead. On dataset B on *inverted index*, for instance, the overhead is less than one millisecond, while the program runs for 13.2 seconds.

Zwift data abstractions turn out to be quite helpful. `ZwiftVec` provides significant space benefits. For example, in dataset A, Zwift saves 99.4% memory space through the use of two-level bit vector for instantiating `ZwiftVec`, compared to the use of regular `vector`

[1]Recall that the experiments on the large datasets use the Spark version on clusters.

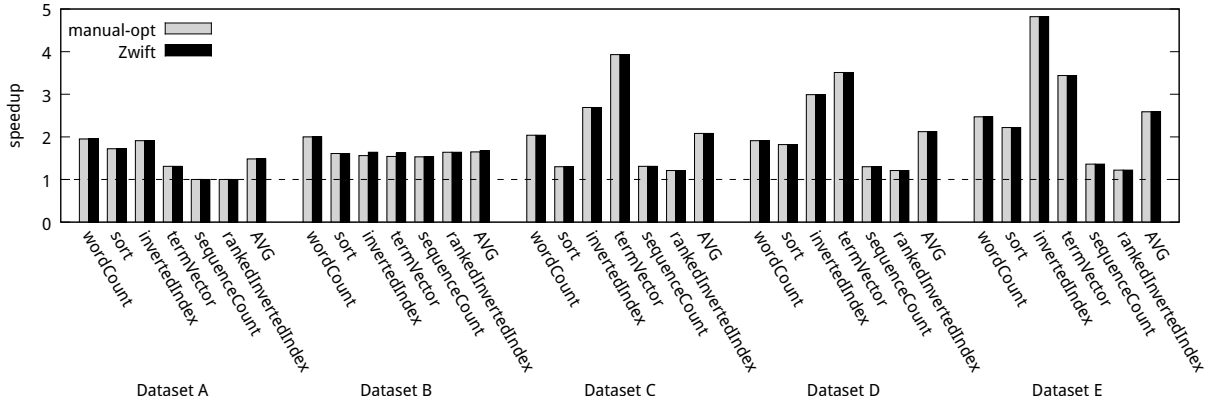

Figure 4: Execution time benefits of speedup over *manual-direct*.

data structure. The reduced memory footprint contributes to better cache and TLB performance. The coarsening of `ZwiftDAG` at loading time also provides benefits. Edge coarsening is enabled for all benchmarks except *sequence count* and *ranked inverted index* due to their order sensitivity. Node coarsening is enabled for *sequence count* and *ranked inverted index* benchmarks. On dataset B, for instance, node coarsening (with a threshold of 100) helps save 5% time over runs without node coarsening.

7.4 Discussion

We discuss the time taken to compress the datasets, its implications, and other aspects of the applicability of Zwift.

In our experiments, a dataset takes Sequitur 10 minutes to 20 hours to compress, depending on the dataset size. The current version of Zwift is designed for datasets that are repeatedly used *without* being frequently updated. In such usage scenarios, the compressed result of a dataset can be used *many times* by different users for various tasks, and the one-time compression time is *not* a major concern. However, if needed, to shorten the compression time, one can employ more efficient parallel or distributed implementations of Sequitur. How to extend Zwift to handle datasets with *frequent updates* is left for future exploration.

Zwift mainly focuses on applications that normally require scanning the entire dataset. Another type of common task involves queries that require random accesses to some locations in a dataset. Such queries are much simpler to support; adding an appropriate index to the Sequitur results could provide benefits. Such tasks can already be supported by other recent techniques (e.g., Succinct [2]), and are hence not included in this paper.

8 RELATED WORK

To our knowledge, Zwift is the first framework and domain specific language designed for text analytics directly on compressed data. There are many other frameworks or languages for specific domains [5, 6, 10, 14, 23, 28, 32, 35]. Some examples include PetaBricks [5] designed for algorithmic tuning, OpenTuner [6] for building program autotuners, Halide [23] for image processing pipeline, HPTA [28] for high-performance text analytics, and TCS [14] for model-to-text and text-to-model transformations.

Deduplication is a topic actively studied in storage systems. Hernández et al. [12] demonstrate the *data merge and purge* problem. String matching [18] and file matching [17] have been studied for deduplication. Sarawagi et al. [25] develop a learning-based deduplication system that focuses on multiple sources. Raman et al. [24] develop an interactive data cleaning system that enables users to build transformations to clean data. As an OS-level technique [27], data deduplication does *not* save processing time in text analytics as TADOC does. If two files contain duplicated content, even if deduplication manages to completely eliminate one copy of the content in storage space, a text analytics application still processes the content *repeatedly* for the two files.

Succinct [2] enables efficient queries on databases with data compression. It is designed mainly for the search and random access of arbitrary strings, rather than supporting general text analytics. TADOC supports much more complex document analytics tasks. For example, *none* of the six analytics problems used in our experiments are amenable to be efficiently supported by Succinct, as they involve complex algorithms and operations beyond search or random access. Moreover, they use different compression methods and internal storage structures. Succinct uses a flat structure, suffix array [19], while Zwift uses Sequitur to create a DAG structure, allowing it to perform complex computations for a wide variety of items.

9 CONCLUSION

This paper introduces Zwift, a new programming framework to enable general programmers to take advantage of TADOC (text analytics directly on compressed data) to save both storage space and processing time in high-performance text analytics. By relieving programmers from most of the efficiency concerns and complexities of implementing text analytics on compressed data, Zwift significantly reduces the effort required by general programmers to apply TADOC while still unleashing TADOC's full power for efficient text analytics. We conclude that Zwift is an effective framework for general-purpose text analytics directly on compressed data, and hope that future research builds on Zwift to demonstrate benefits on even larger data analytics problems.

10 ACKNOWLEDGEMENTS

This work is partially supported by National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61722208, 61472201, 61732014), 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 Beijing Natural Science Foundation (No. 4172031), China Postdoctoral Science Foundation (2017M620992), the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China (No. 16XNLQ02, 18XNLG07), and the open research program of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science (No. CARCH201702). 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.

REFERENCES

[1] Wikipedia HTML data dumps. https://dumps.wikimedia.org/enwiki/, 2017.
[2] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, 2015.
[3] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. PUMA: Purdue MapReduce Benchmarks Suite. 2012.
[4] Amazon. Amazon Elastic Compute Cloud. 2009.
[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI*, 2009.
[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An Extensible Framework for Program Autotuning. In *PACT*, 2014.
[7] Joshua E Blumenstock. Size Matters: Word Count As a Measure of Quality on Wikipedia. In *WWW*, 2008.
[8] Dhruba Borthakur et al. HDFS Architecture Guide. *Hadoop Apache Project*, 2008.
[9] Doug Cutting and Jan Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *SIGIR*, 1989.
[10] Dapeng Dong and John Herbert. Content-aware Partial Compression for Textual Big Data Analysis in Hadoop. *IEEE Transactions on Big Data*, 2017.
[11] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan. Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters. In *ICS*, 2015.
[12] Mauricio A Hernández and Salvatore J Stolfo. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data mining and knowledge discovery*, 1998.
[13] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *New Frontiers in Information and Software as Services*. Springer, 2011.
[14] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006.
[15] Ludovic Lebart. Classification Problems in Text Analysis and Information Retrieval. In *Advances in Data Science and Classification*. Springer, 1998.
[16] M. Lichman. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml, 2013.
[17] Alvaro E Monge, Charles Elkan, et al. The field matching problem: Algorithms and applications. In *KDD*, 1996.
[18] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.*, 2001.
[19] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
[20] Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 1997.
[21] Nicholas A Nystrom, Michael J Levine, Ralph Z Roskies, and J Scott. Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015.
[22] James W Pennebaker, Martha E Francis, and Roger J Booth. Linguistic Inquiry and Word Count: LIWC 2001. *Mahway: Lawrence Erlbaum Associates*, 2001.
[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI*, 2013.
[24] Vijayshankar Raman and Joseph M Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*, 2001.
[25] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive Deduplication Using Active Learning. In *SIGKDD*, 2002.
[26] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, 2013.
[27] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure Data Deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, 2008.
[28] Hans Vandierendonck, Karen Murphy, Mahwish Arif, and Dimitrios S Nikolopoulos. HPTA: High-performance text analytics. In *Big Data*, 2016.
[29] Hao Yan, Shuai Ding, and Torsten Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *WWW*, 2009.
[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 2010.
[31] Uri Zernik. *Lexical acquisition: exploiting on-line resources to build a lexicon*. Psychology Press, 1991.
[32] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. FinePar: Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. In *CGO*, 2017.
[33] Feng Zhang, Jidong Zhai, Xipeng Shen, and Onur Mutlu. Potential of A Method for Text Analytics Directly on Compressed Data. Technical Report TR-2017-4, NCSU, 2017.
[34] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA–Preparatory Data Analytics on Peta-Scale Machines. In *IPDPS*, 2010.
[35] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *MICRO*, 2017.
[36] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 1977.