# Content-aware Compression for Big Textual Data Analysis

Dapeng Dong
MSc

**Thesis submitted for the degree of Doctor of Philosophy**

NATIONAL UNIVERSITY OF IRELAND, CORK

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

May 2016

| | |
|---|---|
| Head of Department: | Prof. Cormac J. Sreenan |
| Supervisors: | Dr. John Herbert |
| | Prof. Cormac J. Sreenan |

# Contents

# List of Figures

# List of Tables

# Notation

| | |
|---|---|
| $H(T)$ | Shannon's information entropy of the given text *T*. |
| $\Theta(\cdot)$ | Indicating the average growth rate of the function. |
| $O(\cdot)$ | Indicating the upper bound on the growth rate of the function. |
| $P_r(T[i])$ | Probability of occurrence of symbol $i$ in text $T$. |
| $\Sigma^{S+}$ | Observed symbols from a given text. |
| $\Sigma^{S-}$ | Non-observed symbols from a given text. |
| $\Sigma$ | Alphabet of a given text. |
| $T$ | A given text. |
| $\varphi$ | Compression ratio. |
| $\lceil x \rceil$ | The smallest integer greater than or equal to $x$. |
| $\lfloor x \rfloor$ | The largest integer less than or equal to $x$. |
| $\propto$ | Proportionality. |

I, Dapeng Dong, certify that this thesis is my own work and has not been submitted for another degree at University College Cork or elsewhere.

*Dapeng Dong*

# Acknowledgements

First of all, I am grateful to my supervisor, Dr. John Herbert, who allowed me the freedom to explore my own ideas, guided me towards research goals, motivated me, and most importantly, kept confidence in me when nothing seemed to progress. You have been a tremendous mentor for me. I am also thankful to my co-supervisor, Prof. Cormac J. Sreenan, for offering advice to my project. I would like to extend my thanks to Prof. John P. Morrison for giving suggestions for the thesis.

I would like to thank my collogues, Dr. Thuy T. Truong, Dr. Xiuchao Wu, Dr. Bingchuan Yuan, Dr. Jason Quinlan, Stefan Mayer, Mohammad Hashemi, and Rezvan Pakdel, for their thoughtful suggestions. I have enjoyed many interesting discussions with the members of the Mobile & Internet Systems Laboratory (MISL) and the Boole Centre for Research in Informatics (BCRI) during my four years of research. I owe much thanks to the MISL administrative manager, Mary Noonan, for her support, and IT staff of the Computer Science department who provided me with equipment for part of my work.

My work would not have been possible without the financial support provided by the Telecommunications Graduate Initiative (TGI) program which is funded by the Higher Education Authority and co-funded under the European Regional Development Fund (ERDF).

Finally, I must express my gratitude to my father, Yanzhi Dong, my mother, Yanling Song, my mother-in-law, Zhenshu Xu, and my wife, Yinhua Fang, who have always supported me in pursuing my interests. Without their love, encouragement, patience, and understanding, this thesis would not have been possible. A special thank you to my son, Runlin Dong, who keeps me moving forward in life during the hardest times.

## Abstract

A substantial amount of information in companies and on the Internet is present in the form of text. The value of this semi-structured and unstructured data has been widely acknowledged, with consequent scientific and commercial exploitation. The ever-increasing data production, however, pushes data analytic platforms to their limit. This thesis proposes techniques for more efficient textual big data analysis with an emphasis on content-aware compression schemes suitable for the Hadoop platform.

In current big data analysis environments, the main purpose of data compression is to save storage space and reduce data transmission cost over the network. Since modern compression methods endeavour to achieve higher compression ratios by leveraging meta-information and contextual data, this context-dependency forces the access to the compressed data to be sequential. Processing such compressed data in parallel, such as desirable in a distributed environment, is extremely challenging. Even though information retrieval systems have developed algorithms that allow operations on compressed data without decompression, they are not sufficiently efficient or flexible in accommodating data updates for big data processing.

This research explores the direct processing of compressed textual data. The focus is on developing novel compression methods with a number of desirable properties to support text-based big data analysis in distributed environments. The novel contributions of this work include the following. Firstly, a Content-aware Partial Compression (CaPC) scheme is developed. CaPC makes a distinction between informational and functional content in which only the informational content is compressed. Thus, the compressed data is made transparent to existing algorithms and software libraries which often rely on functional content to work. Secondly, a context-free bit-oriented compression scheme (Approximated Huffman Compression) based on the Huffman algorithm is developed. This uses a hybrid data structure that allows pattern searching in compressed data in linear time. Thirdly, several modern compression schemes have been extended so that the compressed data can be safely split with respect to logical data records in distributed file systems (Record-aware Compression). Furthermore, an innovative two layer compression architecture is used, in which each compression layer is appropriate for the corresponding stage of data processing (Record-aware Partial Compression). Peripheral libraries are developed that seamlessly link the proposed content-aware compression schemes to existing analytic platforms and computational frameworks, and also make the use of the compressed data transparent to developers.

The compression schemes have been evaluated for a number of standard MapReduce analysis tasks using a collection of public and private real-world datasets. In comparison with existing solutions, they have shown substantial improvement in performance and significant reduction in system resource requirements.

# Chapter 1

# Introduction

Big data is now at the frontier of research, innovation, business, and productivity [JPHS10] [MMCJBB11] [MCB$^+$11]. The term *big data* is understood to be datasets: whose size (*Volume*) is beyond the ability of traditional database software tools to capture, store, and manage; whose rate of production (*Velocity*) exceeds the speed of current systems to process; whose complexity in mixed type and format (*Variety*) challenges existing algorithms to perform efficiently and effectively. These three dimensions [1] of *Volume*, *Velocity*, and *Variety* together characterize big data [Lan01].

## 1.1 The State of Big Data

The value hidden in big data is the driver that pushes research in *big data*. Independent research [MMCJBB11] [Fra15] [AtS15] and much evidence [Joh15] [WVF15] [Ama13] have shown the real-world value of big data. This has led to big data starting to play an essential role in a wide range of applications. Examples include: banking (through analyzing customers' transaction and propensity models [DD13]); healthcare (through analyzing patients' behaviour and sentiment data, clinical data, and pharmaceutical data [PGK13]); education (through collaborative online social media [KRB13]); government (through utilizing new sources of data, engaging public talent, institutionalizing public and private partnerships [Mor15]); social science (through machine learning and causal inference [Gri14]).

As a valuable asset, data is constantly being collected and accumulated from almost all aspects of life. The term *datafication* [2] was coined to capture this phenomenon.

[1]There is also a need to include other dimensions, such as *veracity* which indicates the quality and trustworthiness of source data and data source, and *variability* which denotes the variation in data over time from the same source [ZCJW14], that have been evident from real-world applications by industrial practitioners from diverse backgrounds.

[2]Datafication was first coined by [Ken13]. It is defined as a process of turning all aspects of the world (information) into data.

It is predicted that the global *datafication* process will generate 44 zetabytes of data by 2020 compared to 4.4 zetabytes in 2013 [VTM14]. The social media giant Facebook was producing over 60 terabytes data per day, as reported in 2010 [TSA$^+$10], and Twitter has scaled out their analytical infrastructure from the initial 30 nodes to several thousand nodes, capable of analyzing 100 terabytes of data every day, as reported in 2013 [LR13a]. It should also be noted that a substantial amount of data being collected is semi-structured or unstructured [3], primarily in text format [Rus07]. The sheer *volume* of data and the aggressive speed of growth(*velocity*) are presenting challenges to all aspects of big data analysis from analysis platforms to scalable algorithms. Furthermore, the wide variety of data sources often generate data in various formats at different speeds. Consider the rise of the IoT (Internet of Things) as an example. Billions of different *things* are being connected to the Internet such as pnalersonal wearable devices, smart phones, home appliances, and smart sensors. It is estimated that there are 200 billion *things* on the planet at present, over 25% of them producing data [VTM14]. The IoT creates additional challenges in big data analysis, such as in correlation and regression analysis.

## 1.2 Big Data Development

In order to deal with the challenges brought by big data, there is very active ongoing development of new tools, algorithms, computational frameworks, analytic platforms and deployment provisioning strategies. These provide a basis for solutions to big data analysis. In this section, we discuss current trends toward solutions for big data. We aim to identify the research gaps, propose our solutions and contextualize our work in the field.

- *Scalable algorithms*. The main goals of big data analysis are to extract knowledge and summarize actionable intelligence [CCS12] [FDCD12] from data, and then deliver meaningful results, based on a data-driven approach [ZCJW14] [JGL$^+$14]. Many traditional algorithms in statistical inference, machine learning and data mining are inadequate for big data in terms of parallelism and algorithmic efficiency [MM15]. They are either hard to implement in parallel, for example, Markov chain Monte Carlo (MCMC) simulation for approximating posterior distributions of non-conjugative functions in Bayesian inference, or they need to be redesigned for parallel processing, for example, Matrix Factorization (Collaborative Filtering), Naive Bayes (Classification) and $k$-means (Clustering), as seen in Apache Mahout [Apaf]. There are also developments in approximation and sub-linear algorithms for quickly gaining an insight into big

[3]"*In semi-structured data, the information that is normally associated with a schema is contained within the data, which is sometimes called self-describing*" [Sem97], whereas unstructured data is not organized by pre-defined schemes.

data [WH15].

- *Computational Frameworks and Paradigms*. Currently, computational frameworks developed for big data analytics are mainly designed for processing *stream* or *batch* data. *Stream* processing is designed to capture, analyze, and act on real-time (near real-time or time framed) data streams, such as when monitoring sensors. Widely recognized open-source stream processing frameworks include Apache Storm [Apag], Yahoo! S4 [NRNK10] and Massive Online Analysis (MOA) [BHKP10]. In contrast, *batch* processing is designed for analyzing large datasets offline. MapReduce [DG08], Spark [Apa15] and Dryad [IBY$^+$07] are several frameworks widely used at present. There are also stream-processing enabled versions of MapReduce and Spark.

  Additionally, the paradigm of parallel computation can be split into computation-centric and data-centric aspects. Computation-centric parallelism endeavours to seek better ways of splitting tasks (e.g., Message Passing Interface (MPI) [Ope] [MPI]), whereas data-centric parallelism looks for optimal partitioning of data.

- *Analytic Platforms*. As data volume increases, a distributed file system and storage system are desirable, and analysis based on parallel computing is inevitable. Apache Hadoop [Apad] is the *de facto* dominant, open-source analytic platform currently in use. It provides a unified solution that addresses various aspects of the data analysis life-cycle and includes a distributed file system (HDFS [Apad]), computational frameworks (MapReduce and Spark) and task coordination services (e.g., ZooKeeper [Apaj]). More features are constantly being integrated into the umbrella Hadoop project. Several other analytic platforms exist, such as Microsoft Dryad [IBY$^+$07] and R [The]. In comparison with Hadoop, Microsoft Dryad development has been discontinued and the R framework is primarily focused on statistical computing and graphical techniques. R was not developed for massive parallel data processing and R lacks data life-cycle management in a distributed environment.

- *Deployment Strategy*. Traditionally, the deployment of big data analytic platforms such as Hadoop requires a cluster of servers. An on-site deployment faces scalability constraints. As cloud computing becomes widely available, it has become the main enabling technology for big data analysis due to its dynamic provisioning of resources, cost-efficiency, scalability and elasticity [LQ14] [Col14] [RWZ$^+$15]. Many existing big data solutions are already using cloud computing such as OpenStack Sahara [Sta] and Amazon Elastic MapReduce (EMR) [Ama].

Current solutions attempt to react to big data by scaling up/out analytic platforms [TSA$^+$10] [LR13a] [GSZS14] [RD15] and/or using approximation algorithms [OP15] [BB01]. A question to ask is whether we can do something about the source data. This aspect of leveraging compression for big data analysis in distributed environments has not been given much attention.

> **Scope of this thesis**: Our research sets out to explore new approaches using novel compression schemes for mitigating the challenges for big data analysis created by the unprecedented volume of data. We focus on data-centric batch-processing on textual data using the MapReduce computational framework in the Hadoop data management ecosystem.

## 1.3 Our Approach

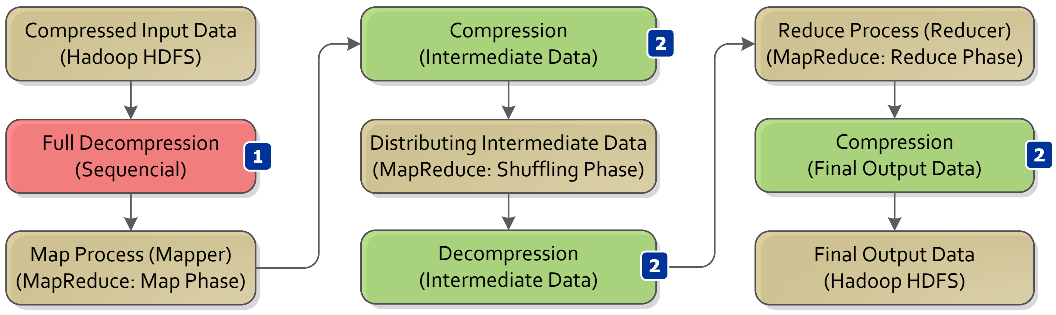

Compressed Input Data (Hadoop HDFS)
Full Decompression (Sequencial)
1
Map Process (Mapper) (MapReduce: Map Phase)
Compression (Intermediate Data)
2
Distributing Intermediate Data (MapReduce: Shuffling Phase)
Decompression (Intermediate Data)
2
Reduce Process (Reducer) (MapReduce: Reduce Phase)
Compression (Final Output Data)
2
Final Output Data (Hadoop HDFS)

Figure 1.1: An illustration of compression as employed by Hadoop and MapReduce.

The main purpose of employing compression in Hadoop is to reduce data size in order to save storage space and lower the data transmission cost over the network. Currently, compressed data cannot be directly involved in MapReduce-based analysis [Whi15]. In order to carry out an analysis, the compressed data must firstly be decompressed fully in a sequential manner, as shown in Figure 1.1 (label 1). This is due to the fact that modern compression schemes often employ data transformation and/or contextualization techniques which create strong dependencies within the compressed data. The data decompression requires the underlying storage system to maintain sufficient free space for holding the decompressed data, and thereafter a MapReduce program can be initiated. Additionally, depending on the compression algorithms, decompression speed varies largely. This can delay considerably the delivery of analysis results when the data volume is large. It should also be noted that previous research has identified that the Input and Output (I/O) system is often the bottleneck for data-centric analysis in a distributed environment [DH15a] [PPR$^+$09] [JOSW10]. We can improve this I/O efficiency by compressing data as much as possible so that the latency created by loading data from persistent storage to memory can be minimized.

> **Research Question 1**: the first question that arises here is, when data is compressed by modern compression schemes such as *Gzip* and *LZO*, whether we can perform decompression in memory as data is being consumed at each computational node of Hadoop (parallel decompression in memory) so that the total decompression time can be reduced in proportion to the number of parallel processes, the data loading time can be kept to a minimum, and the use of extra storage space can be avoided.

If we are to decompress data in memory, in parallel, in a distributed environment, we will have to ensure that the compressed data is splittable and that each data split is self-contained. It also requires maintaining the logical completeness, with respect to the data contents, for each data split. This is because computational frameworks, such as MapReduce and Spark, process data in parallel on a per split basis, independently; each data split is further broken into a group of logical records [4]; a parallel process then deals with a single record at a time.

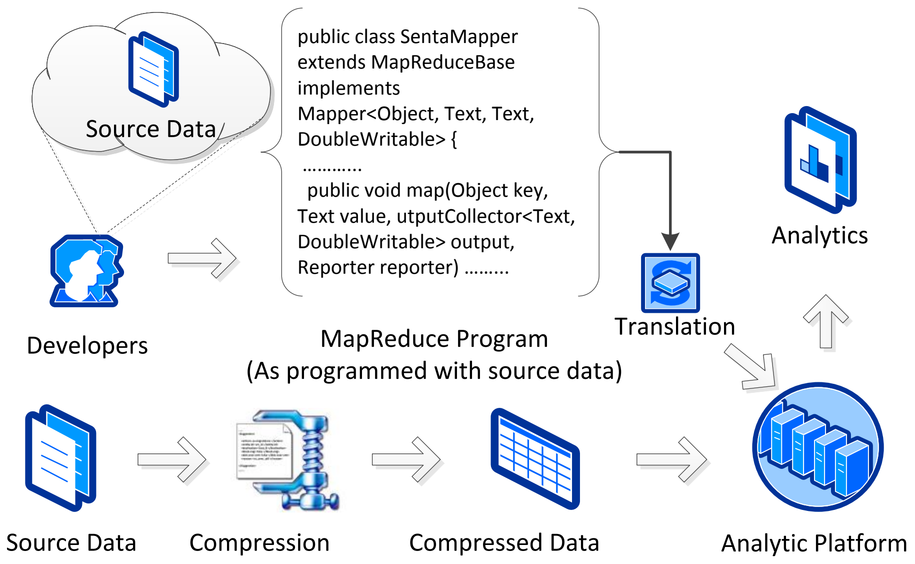
> **Our solution to Research Question 1** is to make the compression process aware of the organization of the data contents, control the use of contextual data, and develop appropriate packaging mechanisms so that compressed data is splittable to the underlying file/storage systems and maintains the logical completeness of each data split to the computational frameworks, without sacrificing much compressibility.

During analysis, intuitively, a MapReduce program will process data in its original format. This is because MapReduce developers can understand the logic and organization of the original data contents to be processed. Also, many existing algorithms and software packages are designed to work with specific data formats, for example, Apache Xerces [Apai] and Apache Commons CSV [Apac] are commonly used software libraries for parsing eXtensible Markup Language (XML) and Comma Separated Values (CSV) formatted documents, respectively.

> **Research Question 2**: the second question is whether we can compress data in such a way that the compressed data can be directly processed in MapReduce without decompression, while ensuring the compressed data is compatible with existing algorithms and software packages as well as transparent to MapReduce program developers.

However, manipulating compressed data poses non-trivial technical challenges, especially in a distributed environment. Traditional solutions to this question are mainly

[4] What constitutes a *record* is data- or application-specific. For example, a record can be a sentence in a plain text file or a row in a database table. This will be further elaborated on in Section 2.1



```
public class SentaMapper
extends MapReduceBase
implements
Mapper<Object, Text, Text,
DoubleWritable> {
 ...........
  public void map(Object key,
Text value, utputCollector<Text,
DoubleWritable> output,
Reporter reporter) ........
```

Figure 1.2: An illustration of the conceptual architecture.

based on indexing techniques [DHL92] [FM00] [WMB99] [CWC+15] [Fal85]. They are developed for information retrieval systems and focus on random access to, and querying of, immutable data content with a cost of high complexity and constraints on being application or domain specific. Since big data analysis often requires manipulating data (for example, extracting columns and re-forming data records), this requires a special way of compressing data so that the element of information (a character or a word) being compressed is independent of its context, thus the compressed data can be freely manipulated. In other words, we need to develop a context-free (referring to [WMB99]) scheme for compression. Although, context-free schemes in general result in moderate compression ratios, the unprecedented volume of big data offers an opportunity for the context-free compression schemes to show their advantages. At the same time, we also need to take several non-functional requirements into consideration including data compatibility, transparency, and accessibility.

- *Compatibility*. Since many existing algorithms and software packages rely on specific data formats and/or pre-defined functional characters to work, we need to compress data in such a way that the original data format and functional contents can be preserved.

- *Transparency*. As data is in compressed format, source information is replaced by codewords. For example, if a developer is to search for a phrase "*big data*" in the source data, they will have in a Java program something like "*contains("big data")*". In order to search for the phrase in the compressed data, they will have to know the corresponding codewords that are used for replacing the phrase in the compressed data. This makes the compressed data opaque to MapReduce developers. We need to bridge this gap between the developers and the compressed data so that the developers can implement analysis logic as if source data is used. This conceptual idea is illustrated in Figure 1.2.

- *Accessibility*. Traditional compression algorithms organize compression model(s) (Section 2.2.2) as a header to the compressed contents. In a distributed environment, data splits are generally processed independently. This requires the compression model(s) to be available to all data splits that belong to the same compressed file. We must separate the compression model(s) from the compressed contents in order to support the requirement on transparency.

**Our solution to Research Question 2** is to develop a context-free compression scheme that can partially compress data in which the informational contents are compressed whereas functional contents are left intact to meet the compatibility requirement, and to provide translation functions as an extension to Hadoop and MapReduce for supporting data transparency and accessibility. Hence, the scheme proposed here is in fact a full solution for Hadoop and MapReduce to work with compressed data rather than a once-off application-specific compression scheme.

Following the work-flow of MapReduce, as shown in Figure 1.1, a Mapper often produces intermediate output data, and this data needs to be temporarily stored in the local hard disks until the MapReduce framework decides to deliver it to the corresponding Reducers for further processing. In order to efficiently distribute the intermediate output data over the network, data can be compressed by one of those general purpose compressors (including *Bzip*, *Gzip*, *LZO*, *Snappy* and *LZ4*) that are currently supported by Hadoop. On arrival, the compressed data is then decompressed and fetched into the Reducers. Eventually, the final output data can be optionally compressed for storage in HDFS. This is shown in Figure 1.1 (Label 2).

**Positive effect of the solution for Research Question 2**. If a MapReduce program can perform analysis with compressed data without decompression, the Mappers will also produce intermediate data in the compressed format, and applying the aforementioned general purpose compression schemes to the compressed data can further reduce the time required for compression as well as data size.

Note that we can achieve high compression ratios by employing customized modern compression schemes, thus data decompression and loading time can be reduced; using context-free compression schemes allows MapReduce programs to manipulate compressed data directly without decompression, thus improving analysis performance and further reducing cost of data transmission over the network.

> **Research Question 3**: the third question is whether we can combine the two approaches to take advantage of both.

The fact is that we cannot apply two different compression schemes to a single dataset at the same time. However, observations have indicated that the first and second research goals can be employed at different stages of the MapReduce processing.

> **Our solution to Research Question 3** is to design a layered architecture so that a MapReduce program can, at different stages, take advantage of the corresponding layers. In other words, we firstly compress data using our context-free compression scheme and then apply our customized modern compressor to the compressed data. As a result, we can achieve data compression ratios close to the *Bzip* compressor and gain a substantial improvement on analysis performance (up to 72%) over using original data.

The research questions together set up the hypotheses for this thesis. Our contributions lie in the solutions to the research questions and their concrete implementations. As a result, we present Content-aware Compression (CaC) schemes suitable for big data analysis in the Hadoop distributed environment.

In addition, cloud computing platforms provide a very suitable environment for parallel processing of big data, and often support Hadoop/MapReduce frameworks. The efficiency and management of the underlying cloud computing environments are concerns that have also been addressed in this research, and solutions have been developed that improve on the pre-existing techniques. This work is described in Appendix E and is not addressed in the main thesis. Improvements in cloud computing efficiency are independent of, and complementary to, the source data analysis improvements. The cloud computing environment solutions can be ported to other analysis frameworks, especially those that are derived from the MapReduce/Hadoop paradigm. In summary, the Content-aware Compression techniques to improve source data analysis efficiency and the management techniques to improve underlying cloud computing efficiency, can separately, and in combination, contribute to better use of resources to cope with the challenge of big data analysis.

## 1.4 Thesis Structure

In this chapter, we have discussed the importance of big data and how it influences many aspect of our lives. We have explained the challenges related to big data and current trends toward solutions for big data analysis. We have clarified our research direction of leveraging novel data compression schemes to mitigate big data issues and contextualized our work in the field. The remaining chapters are organized as follows.

- In Chapter 2, we explain data organization in Hadoop. We then analyze conventional and various ad-hoc compression methods that are important for big data.
- In Chapter 3, we present Content-aware Partial Compression (CaPC) [DH14a] which is our initial attempt towards accelerating textual big data analysis using compression. CaPC focuses on separation of informational and functional contents, and providing complete transparency between analytic platforms, computational frameworks and developers.
- In Chapter 4, we introduce Approximated Huffman Compression (AHC) which was established at an early stage of our research. AHC allows flexible compressed-string searching in linear time using hybrid data structures.
- In Chapter 5, we present Record-aware Compression (RaC) [DH15a]. RaC employs modified higher-order compression methods that can drastically reduce data size while ensuring the compressed data is splittable to the underlying distributed file system. Additionally, RaC is made aware of logical records during compression.
- In Chapter 6, we present Record-aware layered Partial Compression (RaPC) [DH15b]. RaPC uses two different compression schemes that are layer-specific to the corresponding stage of data analysis in MapReduce. RaPC achieves compression ratios comparable to modern general purpose compressors such as *Bzip*.
- In Chapter 7, we present conclusions and discuss future work.

## 1.5 General Convention

Several definitions and conventions are used throughout this thesis as listed below.

- Compression ratio $\varphi$ is given as a percentage and is defined as:

  $$\varphi = \frac{S_o - S_c}{S_o} * 100$$

  where, $S_o$ is source data size and $S_c$ is compressed data size.
- The terminology *source data*, *raw data*, *original data*, and *uncompressed data* are interchangeable.
- Compressed data, by default, indicates compressed textual data, unless indicated otherwise.

- Several compression schemes are introduced in this thesis. We use the term Content-aware Compression (CaC) to refer in general to all these devised schemes.
- In the context of MapReduce, the terms Map output and intermediate data are interchangeable.

## 1.6 Publications

The publications associated with this thesis are listed below.

- The Content-aware Partial Compression (CaPC) scheme for textual big data analysis acceleration in Hadoop was presented at the 6th *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014)*, Singapore, in December 2014 [DH14a].
- The Record-aware Compression (RaC) scheme was presented at the *2015 IEEE International Conference on Big Data (IEEE BigData 2015)*, Santa Clara, CA, USA, in October 2015 [DH15a].
- The Record-aware layered Partial Compression (RaPC) scheme was presented at the 7th *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*, Vancouver, Canada, in December 2015 [DH15b].

In addition to the main contributions given in chapter 3 $\sim$ 6, several studies on efficient management of underlying cloud platforms conducted at an early stage of our research are listed below.

- An efficient server consolidation algorithm for clouds was presented at the 13th *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013)*, Delft, The Netherlands, in May 2013 [DH13b].
- A reference architecture for automation of private cloud management was presented at the 6th *IEEE International Conference on Cloud Computing (CLOUD 2013)*, Santa Clara, CA, USA, in June 2013 [DH13c] and a journal version of this paper has been published by the *International Journal of Cloud Computing (IJCC)* [DH13a].
- A File System as a Service (FSaaS) solution for data management in clouds was presented at the 38th *Annual IEEE International Computers, Software & Applications* Conference (COMPSAC 2014), Västerås, Sweden, in July 2014 [DH14b].

# Chapter 2

# Background Research

*"Some people see the glass half full. Others see it half empty. I see a glass that's twice as big as it needs to be."*

— George Carlin

We have witnessed a rapid evolution of research and development on big data technologies. Important current concerns include efficient algorithms [Apaf] [Spa], parallel computational frameworks [DG08] [Apa15] [Apag] [NRNK10] [IBY+07], comprehensive analytic platforms [Apad] [Clo] [IBMa], scalable deployment strategies [Sta] [Ama], and auxiliary services [Apaj] [HCG+14] [Apae] [Apah] that contribute to an effective big data ecosystem. However, related literature on incorporating compression schemes with big data analysis is limited. Current emerging techniques for compressing big data are mostly ad-hoc approaches optimizing for data-specific and domain-specific datasets. They are not provided as principled solutions. In this context, we firstly analyze how big data is organized in modern distributed file systems, specifically the Hadoop Distributed File System (HDFS), followed by explaining how big data is processed using the MapReduce computational framework [DG08] [Apad]. We aim to identify where and how compression can be used with existing analytic platforms to mitigate big data issues. We then analyze various conventional compression schemes and investigate several important ad-hoc algorithms to discover their potential and limitations for use in big data analysis. We also discuss recent literature on emerging techniques including splittable compression, probabilistic data structures, data deduplication and domain-specific compression. Finally, we summarize the background analysis provided as a foundation to our content-aware compression schemes.

## 2.1 Big Data Organization in Hadoop

Hadoop is a widely adopted data analysis ecosystem. It consists of several components that together provide a platform for managing and analyzing data on a large scale.

A brief history of the development of Hadoop is given in [Whi15]. Many existing big data solutions are based on the Hadoop ecosystem of technologies for example, Cloudera [Clo], OpenStack Sahara [Sta] and IBM BigInsights [IBMa], to name just a few. In this section, we explain how data is organized in the HDFS distributed file system and how data is processed in the MapReduce framework.

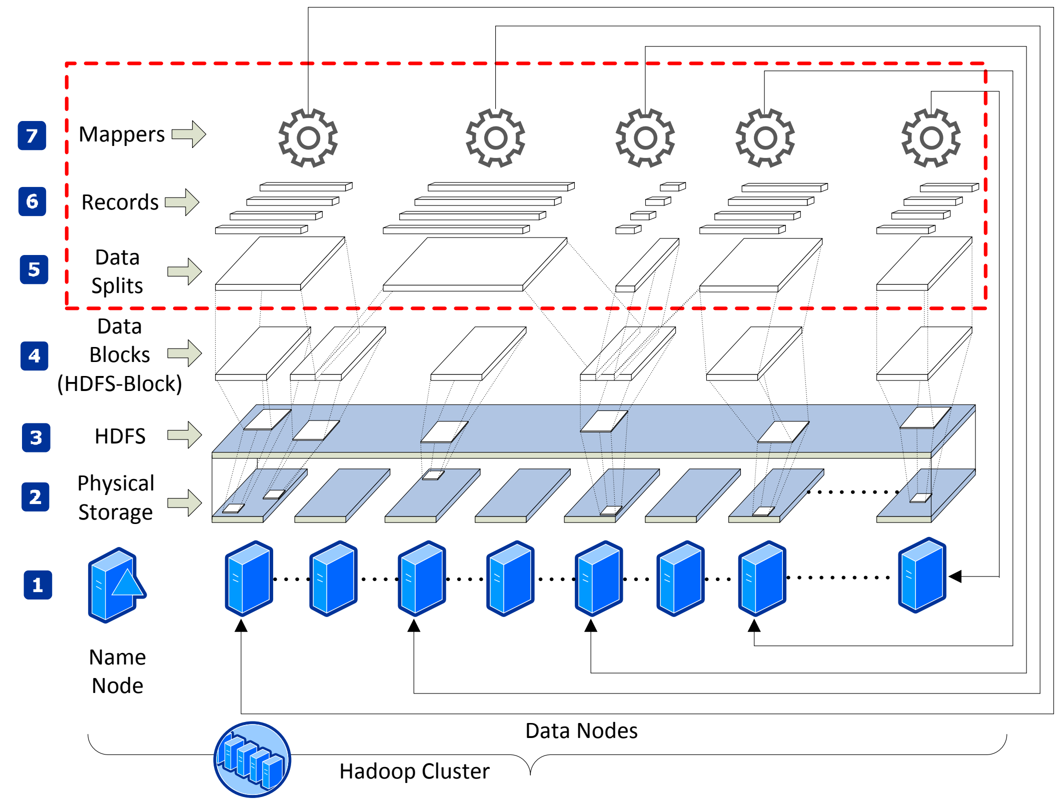


Figure 2.1: Data organization in HDFS and Map processing of MapReduce.

Figure 2.1 illustrates a typical organization of a Hadoop cluster from the physical layer to the MapReduce application layer. At the bottom, a Hadoop cluster consists of a group of physical servers in which one of the servers acts as a controller of the cluster, namely the *NameNode*, and the other servers provide storage space and computation power, namely *DataNodes*, as shown in Figure 2.1 (Label 1). The *NameNode* is responsible for coordinating MapReduce jobs and managing HDFS meta-data. The *DataNodes* are responsible for storing data and providing computation power for MapReduce programs. Each *DataNode* contributes a part of its local storage space (a partition of the local hard disk) to HDFS, as illustrated in Figure 2.1 (Label 2). Thus, HDFS is a virtualized storage system that is comprised of a number of geographically distributed physical hard disks, as shown in Figure 2.1 (Label 3). A file stored in HDFS is split into a series of fixed-size blocks (HDFS-Blocks), as shown in Figure 2.1 (Label 4). HDFS-Blocks are often distributed across *DataNodes*, but are virtually continuous in HDFS. Unlike the traditional organization of storage systems such as NTFS [1] and Ext3 [2] using 4KB data block size, HDFS uses a very large data block size (HDFS-Block size is 128MB by default) chosen for better hard-disk read/write performance. A HDFS-Block is the minimum storage unit in HDFS. It cannot be broken further into smaller pieces.

HDFS only provides a logical view and hides the complexity of organizing data in the underlying distributed storage system. It is an independent service (file system service) provided as a part of Hadoop for data organization and management. Data analysis is carried out by the MapReduce [3] computational framework. As the name implies, a MapReduce program consists of two phases: Map and Reduce. The Map phase operates on a set of key/value pairs. The Reduce phase reduces the outputs from the Map phase by values which share the same key. However, the Map phase can also perform the Reduce logic locally, thus there is no clear separation of responsibility for the Map phase and Reduce phase. It may be better to understand the MapReduce paradigm by an intuitive example. For instance, given a job of finding the maximum value(s) in a large financial report dataset, the dataset will be firstly broken into a series of data blocks, and each data block will be processed independently by a dedicated Map process (Mapper, as indicated in Figure 2.1 <Label 7>) to find the *local* maximum value(s). The results from all Mappers will then be aggregated to a Reduce process (Reducer) to find the *global* maximum value(s).

As described, each Mapper will be assigned to a HDFS-Block. Within a HDFS-Block, data will be further broken into logical records (this depends on the dataset, for example a logical record can be a sentence in a text file or a row in a database table), as indicated in Figure 2.1 (Label 6). The record parsing process is done by the MapReduce *Record Reader* component. A Mapper will process a record at a time. However, HDFS simply splits data by size. It is very likely that a HDFS-Block will contain incomplete records at the boundaries. To ensure record completeness, MapReduce further organizes HDFS-Blocks into *data splits*, (Figure 2.1 <Label 5>). Essentially, a *data split* is a logical view of a data block which is guaranteed to contain a set of complete records as shown in Figure 2.1.

Intermediate output data from each Mapper is firstly cached in an in-memory buffer, as indicated in Figure 2.2 (Label 1). When the utilization of the caching buffer reaches a certain threshold, all cached data in the buffer will be materialized to persistent storage (local hard disks). This process is called *Spilling* in the context of MapReduce. Data in a *spill* will then be sorted by keys and partitioned according to the number of Reducers

[1]NTFS stands for Microsoft® New Technology File System. NTFS has default 4KB block size for storage volume less than 16TB on Windows® 7 or above.

[2]Ext3 is Linux Third Extended File System. It commonly uses 4KB block size to address 16TB storage volume.

[3]http://hadoop.apache.org/. A comprehensive introduction to MapReduce can be found in [MVE+14].

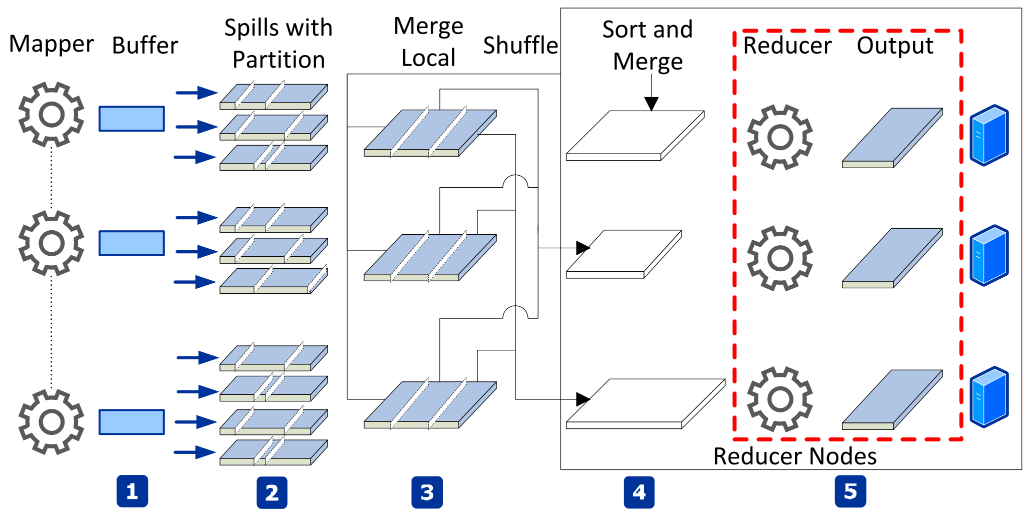


Figure 2.2: Data organization in HDFS and Reduce processing of MapReduce.

configured for the job, as indicated in Figure 2.2 (Label 2 and 3). Upon completion of the Map processes, each sorted partition will be distributed to a corresponding Reducer over the network. At each Reduce computational node (the physical server that runs the Reduce phase of the program), the received partitions will be sorted again and merged into a single data block, and then fetched to the Reducer, indicated in Figure 2.2 (Label 4). The final results (Reduce output data) will be stored in HDFS eventually, as indicated in Figure 2.2 (Label 5).

Questions arise when using compressed data in Hadoop.

1. If source data is compressed using general purpose compression schemes such as *Gzip* [Gzi] and *Bzip* [Bzi], after splitting the compressed data into HDFS-Blocks, can we ensure that each HDFS-Block is still independently de-compressible?
2. If data is in compressed format, how can we organize HDFS-Blocks in *data splits* and guarantee record completeness?
3. If data is in compressed format, can we break a *data split* into records efficiently?
4. Can we process data in compressed format without sacrificing performance?
5. How can we distribute the Map output data (intermediate data) to corresponding Reducers more efficiently?

These questions set requirements on compression schemes suitable for big data analysis in Hadoop. The criteria we focus on are whether the compressed data is splittable, directly consumable [4], and aware of source data contents. In the following sections, we

[4]There are two scenarios in big data analysis. In the first case, data will be analyzed by existing algorithms without modification (read-only) throughout the analysis. In the second case, data will be

discuss related work by examining existing conventional and random access compression algorithms for textual data, and recent emerging techniques specifically designed for dealing with big data.

## 2.2 Conventional Compression Schemes

Data compression originated in communication systems. Since the invention of the Morse code [ITU09] in the early nineteenth century to the development of information theory [Sha48] and several subsequent important compression related coding schemes [Sha49] [Huf52] in the middle of twentieth century, the research and development on data compression had developed slowly due to absence of demand. The rise of the Internet and information explosion prompted a rapid development of data compression methods from the simple *LZ*77 (Lempel-Ziv) [ZL77] in 1977 to the state-of-the-art *LZMA*2 (Lempel–Ziv Markov Chain) [Igo15] at present. Over the years, numerous general purpose and specialized data compression methods have been developed. It is infeasible to examine all of them. In this section, we examine several important compression techniques which are often used as building blocks for modern compressors. As a typical compression scheme often consists of three phases: *Transformation*, *Modeling*, and *Encoding*, we discuss commonly employed techniques for each phase. A representative algorithm is thereafter selected to further evaluate the benefits and limitations of their use in the context of big data. The emphasis is on textual data related compression methods.

### 2.2.1 Transformation

Transformation is selective and often used as a preprocessor for compression. It involves reordering data. It does not compress data. But it leads to interesting results where the same characters or a similar set of characters tend to be grouped together. The repetitiveness can then be captured by subsequent modeling and coding schemes, and thereby it supports higher compression ratios. Generally, transformation is performed on a per data block basis, thus it is also known as block sorting. There are several common methods for data transformation operating at the character level including Burrows-Wheeler Transform (BWT) [BW94], Symbol Ranking [Fen97] and Lexical Permutation Sorting Algorithm (LPSA) [AM97]. Data transformation has also been extended to word level [IM01] with extra cost in terms of complexity on parsing words from a given text, but based on the same principles. Among these transformation techniques, BWT is one that is widely used, such as in the well-known *Bzip* [Sew00]. We use BWT as a case study to further examine the suitability of using transformation

modified and processed, for example, extracting columns and re-forming database records. In the context of this thesis, if compressed data is directly consumable, it must fulfill the requirements for both scenarios.

techniques for compressing big data with regards to splittable and directly consumable data.

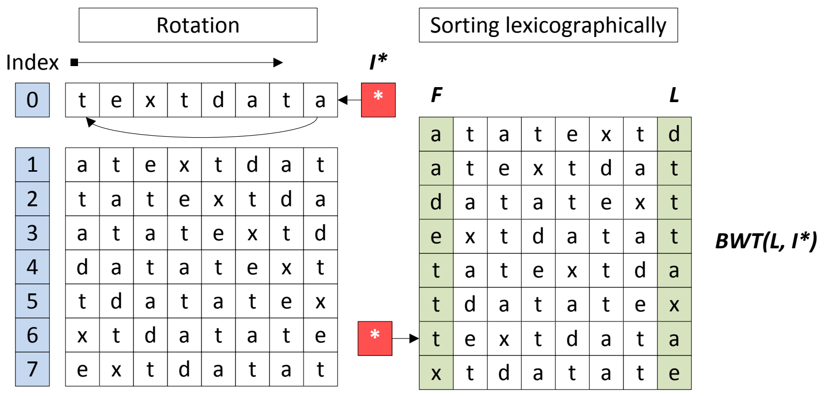


Figure 2.3: An example of Burrows-Wheeler transformation.

The BWT transformation starts by taking a block of data in a row. Imagine we fit the block of data into a cyclic queue and move the first element of the queue to the end of the queue until all of the characters in the block have been moved once. For example, if we are trying to perform BWT transformations on the given text "*textdata*", by recording the rotation steps, we will have a matrix generated for the text "*textdata*" as illustrated in Figure 2.3 (*left*). When we sort the matrix by the first column in a lexicographical order, we will have a second matrix shown in Figure 2.3 (*right*). This rotation and sorting do not shrink data size. It produces an interesting result. If a symbol occurs frequently in the block, after sorting the first column $F$, the prefix character of the symbol which is now in the last column, $L$, tend to group together. This result makes other compression methods, for example, Move-To-Front (MTF) coding [BSTW86], more effective. Most importantly, the transformation process is reversible by only knowing $L$ and the starting index $I^*$.

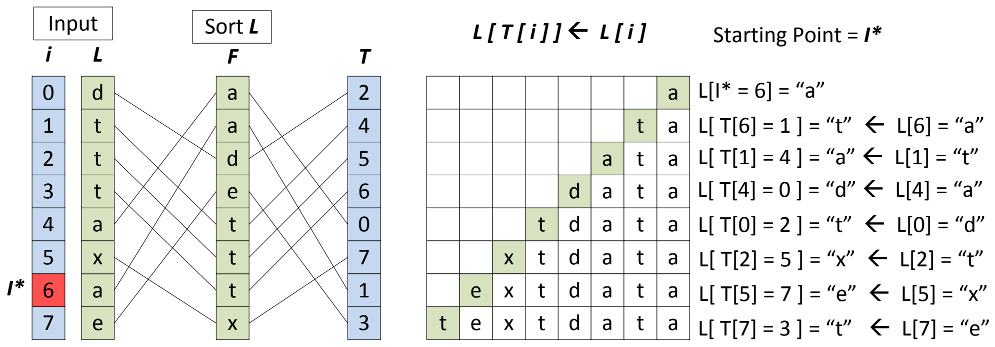

Input
Sort L
i
L
F
T
I*
L[T[i]] ← L[i]
Starting Point = I*
L[I* = 6] = "a"
L[ T[6] = 1 ] = "t" ← L[6] = "a"
L[ T[1] = 4 ] = "a" ← L[1] = "t"
L[ T[4] = 0 ] = "d" ← L[4] = "a"
L[ T[0] = 2 ] = "t" ← L[0] = "d"
L[ T[2] = 5 ] = "x" ← L[2] = "t"
L[ T[5] = 7 ] = "e" ← L[5] = "x"
L[ T[7] = 3 ] = "t" ← L[7] = "e"

Figure 2.4: An example of Burrows-Wheeler inverse transformation.

The inverse process firstly sorts $L$ lexicographically. The outcome is in fact the column $F$ we have seen during the transformation phase. Once we have $L$ and $F$, we need to build a transformation index vector $T$ that indicates the relationship between $L$ and $F$. The relationship is as follows. The $i^{th}$ value of $T$ is the index of the $i^{th}$ character of $L$ in $F$, as illustrated in Figure 2.4 (*left*). After calculating vector $T$, the original message can be retrieved from $L$ using the relationship defined by $L[T[i]] \Leftarrow L[i]$ in reverse order. The last character of the original message is given by $L[I^*]$. In this example, $I^*$ = 6, this indicates that the last character of the original text is the $6^{th}$ character in the column $L$, which is the character "*a*" (note that the index of $L$ starts from zero). This is illustrated in Figure 2.4 (*right*).

Given a block size $n$, the rotation phase takes $O(n)$ time and the sorting process requires as least $O(n \cdot log(n))$ time assuming we use quick-sort or merge-sort algorithms. A bigger $n$ potentially improves the compression ratio, but it consumes more time. A compression scheme that employs block-sorting techniques needs to consider this overhead. For example, *Bzip* uses BWT as a preprocessor. In comparison with *Gzip*, *Bzip* is much slower both in compression and decompression speed but with slightly higher compression ratios. In principle, a BWT enabled compressor can produce a series of self-contained outputs. By knowing the block size, the compressed data is splittable to the HDFS. However, subsequent compression processes determine whether inter-block connections are created. In addition, the permutation of characters (rotation and sorting) prevents random access to information in the internal blocks without special indexing techniques as will be explained in Section 2.3.3.

### 2.2.2 Modeling

Generally, a text compressor is either an implementation of a statistical or dictionary method. Statistical methods are based on the principle that a symbol with a higher probability of occurrence can be represented by fewer bits subject to Shannon's theorem [Sha48]. It is also referred to as *symbolwise* compression [WMB99]. In digital computers, compression aims to use a minimum binary alphabet sequence to represent a given message. In information theory, if a message $X$ consists of $n$ independent symbols [5] $\{x_1, x_2, ...x_n\}$, the probability of occurrence of each symbol $x_i$ in $X$ is given by $Pr(x_i)$, then the expected number of bits needed for $X$ is given by Equation 2.1.

$$H(X) = -\sum_{i=1}^{n} Pr(x_i) log_2 Pr(x_i) \tag{2.1}$$

In the context of statistical compression, the function of a model is defined as:

[5]Depending on the context, a symbol can be a character, a group of characters, or a word.

*"The function of a model is to predict symbols, which amounts to providing a probability distribution for the next symbol that is to be coded."* [WMB99]

There are three basic categories of modeling techniques: static, semi-static, and adaptive. A static model is built based on prior knowledge or experience gained from similar data. It is often a subjective choice. For example, a static list of characters can have pre-assigned probability for each character. Using static models often results in poor compression ratios and is rarely used by modern compressors. However, static models can be useful when the data contents are known or well-defined. For example, social media data such as Facebook posts often contain commonly used English words. A simple static word list can be an effective model for such type of data. Especially for big datasets, constructing model(s) on the fly can be very time consuming.

In contrast, building semi-static models requires scanning through data contents, hence an accurate symbol probability distribution table can be constructed. This implies that compression schemes based on semi-static models require two passes over the data. The first pass is used to build a model(s) and the second pass carries out the actual compression. There are two basic issues with the use of semi-static models. Firstly, in the big data context, traversing large datasets twice can be extremely inefficient. Secondly, when data size becomes larger, potentially, the probability distribution of symbols become smoother. This implies that codes for symbols (assume using character-based compression) tend to have length similar to the standard ASCII codes, thus resulting in poor compression. To overcome these issues, many applications [Sew00] [lGA] [Igo15] [FMMN04] [ZL77] [Deu96] employ block-wise compression. Dedicated models are built for each block of data. It is not necessary for each block to have the same size. For example, the *Deflate* algorithm [Deu96] uses two dedicated Huffman trees [Huf52] (compression models built from the Huffman algorithm) for encoding symbols of a single data block. The size of each data block is decided by the usage of the internal buffer and the efficiency of the associated Huffman trees. Nevertheless, the variable-length input data blocks result in variable-length compressed output data blocks, thus splitting compressed data in HDFS requires the recording of block boundaries. It is even more challenging to form higher level *data splits* for MapReduce. Additionally, using multiple models for a single dataset makes accessing compressed data very problematic in distributed environments. This will be further elaborated in the following chapters.

An adaptive model [6] starts with a flat probability distribution. For example, considering an arbitrary text file that contains standard ASCII codes and extended ASCII codes, such as multi-language articles, there are in total 256 characters, and each character will be given a probability of $\frac{1}{256}$ initially (there are other ways for the initial probability

[6]Adaptive models and adaptive coding should not be confused. An adaptive model evolves with new contents. Adaptive coding implies changes caused by new inputs to a model at a sender can be dynamically updated in the model at a receiver [LR81], as used in data communication systems.

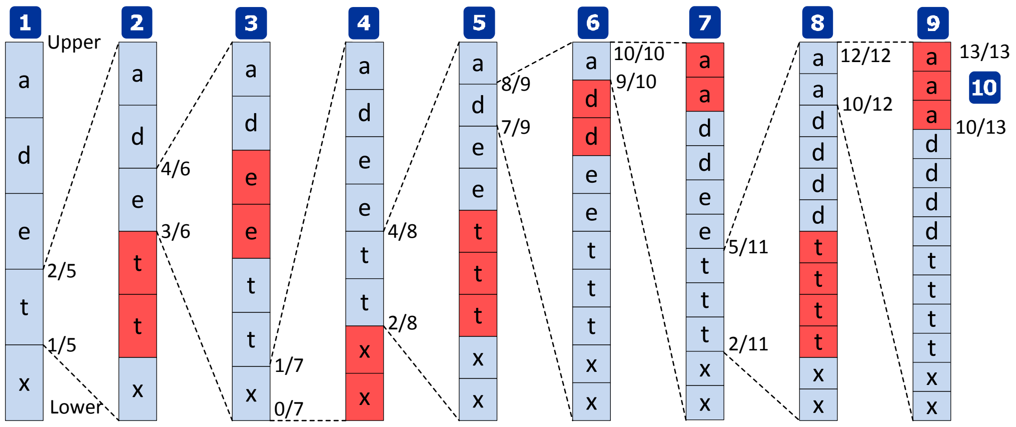
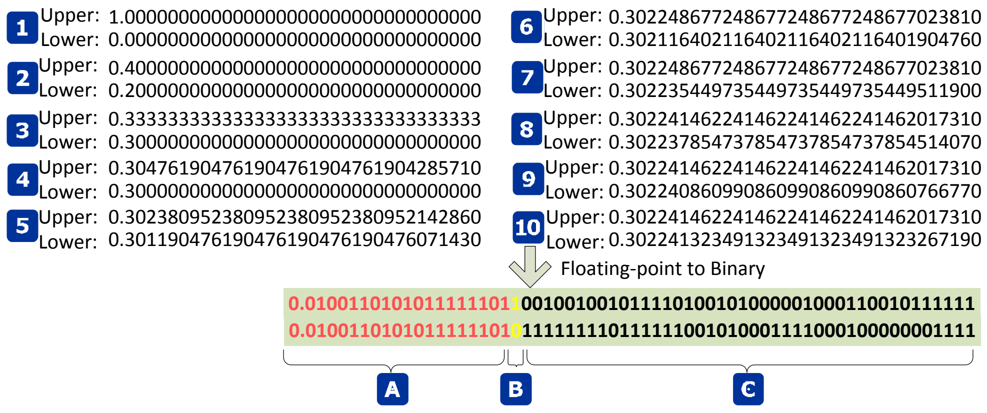


Figure 2.5: An example of adaptive arithmetic coding step-by-step.



```
1  Upper: 1.00000000000000000000000000000000     6  Upper: 0.302248677248677248677248677023810
   Lower: 0.00000000000000000000000000000000        Lower: 0.302116402116402116402116401904760
2  Upper: 0.40000000000000000000000000000000     7  Upper: 0.302248677248677248677248677023810
   Lower: 0.20000000000000000000000000000000        Lower: 0.302235449735449735449735449511900
3  Upper: 0.33333333333333333333333333333333     8  Upper: 0.302241462241462241462241462017310
   Lower: 0.30000000000000000000000000000000        Lower: 0.302237854737854737854737854514070
4  Upper: 0.304761904761904761904761904285710     9  Upper: 0.302241462241462241462241462017310
   Lower: 0.30000000000000000000000000000000        Lower: 0.302240860990860990860990860766770
5  Upper: 0.302380952380952380952380952142860     10 Upper: 0.302241462241462241462241462017310
   Lower: 0.301190476190476190476190476071430        Lower: 0.302241323491323491323491323267190
                                                     Floating-point to Binary
               0.01001101010111111011 1 001001001011110100101000001000110010111111
               0.01001101010111111010 0 111111110111111001010001111000100000001111
               A                      B C
```

Figure 2.6: An example of adaptive arithmetic model evolution.

assignment [WMB99]). Regardless of what the first character is in the given text, the estimated probability for the first character is given by $\frac{2}{257}$. Based on Shannon's theorem as given in Equation 2.1, we can calculate the minimum number of bits needed to encode the first character. When compression moves forward in the text file, the model evolves accordingly. This is also known as an order-zero model. Adaptive Arithmetic coding [LR81] and Adaptive Huffman coding are two widely used schemes based on order-zero models. We use Adaptive Arithmetic coding to further examine the potential and limitations of using order-zero models with respect to splittable and random access in big data analysis.

Using the same example given in Section 2.2.1, in order to encode the message "*text-data*" using an Adaptive Arithmetic coder, we assume the alphabet {*a*, *d*, *e*, *t*, *x*} is known. Then, we need to assign an initial probability to each character with equal weights as shown in Figure 2.5. The algorithm treats the message to be encoded as a stream. When a new character is encountered, the probability assignments for all characters in the model are adjusted accordingly as shown in Figure 2.6 (*step* <1 – 10>). For each step forward, the previous probability interval indicated by upper and lower bound will be proportioned by the evolving model. For example, "$t$" is the first character in the given message. Initially, a flat probability of 0.2 is given to each character (five distinct characters in the alphabet share the entire interval [0, 1], and are sorted in lexicographical order). The probability given to character "$t$" is allocated an interval [0.2, 0.4]. Moving forward, the second character is "$e$". Within the interval assigned for the first "$t$", the new probability distribution for "$e$" is then calculated as the interval assigned for "$t$" divided by six since the character "$t$" was observed before "$e$". Thus, the new interval is [0.3, 0.33] as shown in Figure 2.6 (*step 3*). The process moves on until the last character is reached. This implies that message decoding must start from the beginning of the compressed message in order to identify the initial interval. When the compression reaches the last character, the final interval becomes very narrow (depending on the length of the message) and constrained by upper and lower bounds in real numbers. In fact, these real numbers in decimal can be very long. They do not provide any obvious compression. To achieve compression, these real numbers are converted into binary real numbers [7] as shown in Figure 2.6 (*step 10*). In principle, any number between the final upper and lower bound can be used as the compression result. Therefore, it is unnecessary to use the full length of the binary real numbers. Just the common part of the upper and lower bound (as indicated in Figure 2.6 by <Label A>) plus a single bit in difference (as indicated in Figure 2.6 by <Label B>) are sufficient to decode the original message. The remaining part of the binary real number (as indicated in Figure 2.6 by <Label C>) can be ignored. From this example, we can see that the Adaptive Arithmetic coding is very efficient as it uses only 20 bits to encode 8 characters (64 bits in the ASCII encoding scheme). However, due to the stream-based processing, we cannot split the compressed data nor have random access to it. Thus, a common shortcoming when using an adaptive model is lack of random access. Further explanation and variations of Arithmetic coding can also be found in [Sal08] [WMB99] [Ris76] and [MT02].

There are higher-order models based on various prediction techniques such as finite-context, finite-state and neural networks. Prediction by Partial Matching (PPM) is a representative algorithm [CW84] that uses finite-context prediction. PPM maintains several models from lower order (order-zero) to higher order (order-4 is common). A higher order model uses longer contextual data. For example, using the same message "*textdat**a***", an order-zero model calculates probability for the last character "$a$" by counting how many "$a$" occurred previously, an order-1 model counts how many "$ta$"

[7]Floating-point to binary arithmetic is defined in IEEE 754 standard (the most recent version is defined in IEEE 754-2008). http://grouper.ieee.org/groups/754/

were observed. Probability of symbol occurrence is estimated by all models and the highest value is selected for encoding. This is due to the fact that higher probability results in lower information content subject to Shannon's theorem. Similarly, Dynamic Markov-Chain (DMC) [CH87] uses finite-state and PAQ [8] [Mah05] employs neural networks for predicting symbol probability. A final point should be noted for statistical models. Static and semi-static models are context-free models. They are different from order-zero models. A context-free model is non-predictive whereas an order-zero or higher-order model is predictive. These adaptive models make great use of contextual data and this makes random access to the compressed data much harder.

Dictionary methods are also commonly used for textual data compression. They achieve compression by eliminating repetitiveness and achieve greater compression ratios by leveraging contextual data. Many existing state-of-the-art compressors are based on dictionary methods. For example, *Gzip* [lGA], 7-*Zip* [Igo15], Lempel-Ziv-Oberhumer (*LZO*) [Mar15] and Roshal Archive compRessed (*RAR*) [RAR15]. They are ultimately derived from *LZ*77 [ZL77] and *LZ*78 [ZL78]. *LZ*77 is one of the earliest dictionary methods, developed in the 1970's. The algorithm works by maintaining a context buffer (*Sliding Window*) which contains the recent input stream (source format) that has already been encoded. It works by finding the longest immediate succeeding string starting at the current compression position for a match in the context buffer, and replaces it by a triple which comprises of the starting position in the context buffer, length of the matched string and the immediate succeeding character of the matched string in the current compression stream. *LZ*78 [ZL78] and its variant Lempel–Ziv–Welch (*LZW*) [Wel84] maintain an in-memory dictionary which contains strings that have been observed so far. A new string that is found in the dictionary is replaced by a pointer to the index of the match in the dictionary. The use of context information causes the same problems as decoders can only decode compressed contents sequentially. This means that splitting compressed data requires recording boundaries of compressed blocks (non-stream based). Additionally, the size of the context buffer varies depending on the algorithm. For example, *LZ*77 can use an arbitrary size for the context buffer (64MB or even bigger). *Deflate* uses a 32KB/64KB context buffer. The longer buffer potentially yields better compression ratios, but it makes the dependencies between compressed blocks stronger and subsequently makes splitting compressed data harder for HDFS.

In contrast to context-dependent compression, a simple dictionary-based compression method is described in [Sal08]. The algorithm is a two-pass compression scheme. It starts by calculating probability distributions for each ASCII character ($0 \sim 255$) from a given text in the first pass, and sorts characters by probabilities in descending order.

[8]PAQ is the name of an experimental open source project working on specialized compression algorithms that aim to achieve the highest compression ratio possible while ignoring constraints on space and time complexity.

The compressor then reads the input stream again, and replaces each character by its corresponding index in the probability distribution table and prepends three extra bits to indicate the length of the index. This method is simple and intuitive. The problem is that when the input data size is big, going through the data twice leads to poor compression speed, for example, at terabyte, petabyte or even exabyte scale. Also, the compression ratio of the scheme is poor and so that makes it much less interesting for big data.

### 2.2.3 Encoding

Huffman coding [Huf52] and Arithmetic coding [LR81] are the two most widely used encoding schemes. Both can produce optimal codewords. They are often referred to as *entropy* encoders [WMB99] [Sal08]. Arithmetic coding creates dependencies along the compression stream regardless of whether a static, semi-static or adaptive model is used. In contrast, Huffman coding with a static or semi-static model generates prefix-free codewords for symbols, and codewords are independent.

The basic task of Huffman coding is to generate shorter codes for more frequently occurring symbols in a given message. The algorithm takes the probability distribution of symbols as an input and starts by constructing a Huffman tree. Each symbol is treated as a leaf in the tree. The construction process starts by pairing lowest probability symbols in a recursive manner until all symbols are connected in the tree. Given the same input (message and model), Huffman codes are not unique. For example, if two symbols have the same probability, which symbol is placed on the left/right branch of the tree will result in different codewords. To avoid ambiguity in decoding, the Huffman tree construction process needs to agree on some rules. For example, the left branches of the tree are always labelled with 0 and 1 for the right, or vice versa. For symbols having the same probability, symbols with lower values (e.g., converted to ASCII code values) are placed on the left or vice versa. The decoder reads bit by bit from the compressed data to follow the branches in the Huffman tree until a leaf is reached. Then it can be safely assumed that the bit pattern was used for encoding the symbol stored in the leaf and the decoder emits the corresponding symbol to the output.

As mentioned above, the Huffman coding algorithm produces independent codewords. Codewords have variable-length measured in bits. But, starting at a random position in the compressed data, it does not indicate how to determine boundaries between codewords. It works only when the decoding proceeds in order from the beginning to the end. This implies the Huffman coding does not allow random access unless we can develop techniques that can effectively detect codeword boundaries such as indexing, or package Huffman compressed data in a specific format. Further explanation and solutions for using Huffman compressed data in Hadoop and MapReduce are given in Chapter 4.

Huffman coding can work with characters. It can also work with words in a different way. The method is called Canonical Huffman coding [HL90]. Instead of using characters for leaves, words are used. Given the probability distribution of words, the conventional Huffman algorithm is used to calculate the length of codewords for words. Words are then sorted and partitioned by the length of their corresponding codewords. Within each partition, words are sorted in alphabetical order. A prefix-free code is given to the first word in each partition and codewords for the following words in the same partition are given by incrementing the first codeword of the partition [WMB99]. This makes the algorithm very efficient for decoding, because codewords and their corresponding words can be quickly identified from a lookup table. However, Canonical Huffman coding produces variable-length bit codes. This requires all information in a given text to be coded including words and non-words (e.g., punctuation characters). Considering that Internet generated data often contains Unicode contents, how to split words and build a complete word-to-code map is a problem. Several other coding schemes [Sha49] [FK96] [Mar79] are also available, but less commonly used for textual data compression. They present similar issues as described for Huffman and Arithmetic coding.

## 2.3 Random Access Compression Scheme

Random access on compressed data is an active topic in the field of information retrieval. It allows one to obtain the original records at random positions and offers a certain degree of freedom in terms of content manipulation without decompression. Common approaches are based on context-free compression, self-synchronizing codes, and indexing techniques.

### 2.3.1 Context-free Compression

Context-free compression uses various coding schemes with static or semi-static models/dictionaries. The main driver for this approach to be successful is codeword independence. In this thesis, we classify context-free schemes into three categories as shown in Figure 2.7. A bit-oriented/word-based scheme is characterized by relatively higher compression ratios, byte-oriented schemes achieve better performance on compressed string matching, whereas character-based schemes support searching for arbitrary strings.

In the early development of context-free compression, byte-oriented word-substitution approaches [Man97] [SdMNZBY00] [Cra99] were common. [Man97] introduces a scheme that uses a single byte drawn from the extended ASCII code pool to replace a pair of characters (digram) in a given text. The code space is limited to 128 as that

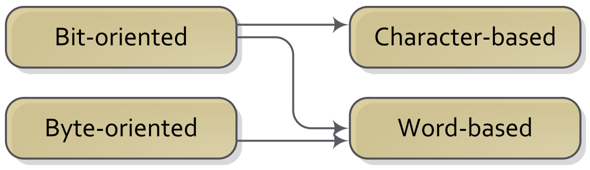


Figure 2.7: Classification of context-free compression schemes.

is the maximum number of characters available in the extended ASCII table (128 ∼ 255). The advantage of the scheme is that it allows string searching directly on the compressed data. The drawbacks are that:

- it requires scanning through the entire dataset(s) to find appropriate digrams,
- it is difficult to identify most frequently occurring digrams.

Thus, the algorithm is only suitable for data with reasonable size and the achievement on compression ratio is relatively low. If the source data contains Unicode characters, the algorithm for decompression may fail due to the fact that the Unicode scheme employs extended ASCII codes. A similar word-based compression using Huffman coding combined with a semi-static model was proposed by Silva et al. [SdMNZBY00]. Recall that Huffman coding produces variable-length codewords (bit-oriented). Decoding must start from the beginning of the compressed data. To overcome this problem, the authors [SdMNZBY00] use byte-oriented Huffman coding. Each codeword is guaranteed to have a whole number of bytes in length. Furthermore, the Most Significant Bit (MSB) of each byte is used to indicate whether this is the beginning of a codeword. This mechanism sacrifices some leading and trailing bit(s) to allow random access. But the limitation is clear. The irregular use of the extended ASCII codes (setting MSB to one) makes it difficult for these aforementioned schemes to deal with Unicode contents. Most importantly, the current version of the MapReduce framework only supports standard ASCII and UTF-8 encoding schemes for strings. When a MapReduce program reads textual data from HDFS, it automatically converts text into UTF-8 strings by default. This will very likely cause *Invalid Format* errors during the conversion processes, because the use of extended ASCII codes does not follow the rules defined by UTF-8. Re-Pair [LM99] is another byte-oriented compression scheme. The authors [LM99] use a single byte-symbol to replace the most frequently occurring character-pairs. The process iterates until no more common pairs can be found. Because of the iteration process, previously generated byte-symbol(s) are also treated as character(s) which can be treated as a part of a new pair in a subsequent iteration process and this creates embedded dependencies which prevents random access to compressed data.

For many domain specific datasets, splitting data into words can be difficult, for example, in a genome sequence or financial report. In contrast, a character-based scheme provides flexibility for arbitrary string searching. Ternary Digit [Kat12] is a character-based, bit-oriented scheme. It uses a ternary digit [9] and binary 11-pair scheme to generate variable-length codewords. Codeword assignment is based on the probability of occurrence of characters in a given text. It is based on the same principle as used in Huffman coding. Thus, this compression method presents the same issues that have already been identified for Huffman coding. Similarly, Canonical Huffman coding [HL90] (bit-oriented) is often used for word-based compression. As per described in Section 2.2.3, Canonical Huffman provides faster decoding speed using lookup tables. However, the variable-length encoding scheme poses the same problem for splitting compressed data. Several other ad-hoc algorithms such as grammar-based [BLR$^+$11] [AL98] and compressed string matchings [BDM13] [FT98] [WM92] [BFG09] are specific for random access to static contents. Generally, grammar-based compression looks for compact grammars that can be used to represent source messages. However, depending on data contents, finding compact grammars for arbitrary messages is a Non-deterministic Polynomial-time hard (NP-hard) task. Compressed string matching often employs *succinct* data structures [Jac88] and indexing techniques which create strong dependencies between data contents, thus making splitting compressed data non-trivial. Additionally, in big data analysis, especially in MapReduce programs, manipulating data is common. We need more than just random access.

### 2.3.2 Self-synchronizing Codes

Recall that character-based schemes are flexible for arbitrary string searching. The per-character encoding needs to employ bit-oriented approaches in order to achieve compression. In fact, many existing coding schemes such as Huffman coding [Huf52] and Arithmetic coding [HL90] are variable-length and bit-oriented. On the other hand, most modern operating systems, software and applications manipulate data at byte-level. In HDFS, data is split into a number of bytes (128MB by default). This poses a concern in that splitting compressed data will very likely create a broken code at the splitting point. For example, the last byte of the first HDFS-Block and the first byte of the immediate succeeding HDFS-Block may each contain a partial codeword for a single character. In the worst case scenario, the second HDFS-Block can be *successfully* decoded, but produce completely different erroneous information from the source data (error propagation).

Data communication and information retrieval systems use synchronization points and self-synchronizing codes to solve the broken code problem [WMB99]. A synchronization point can be a single byte or a sequence of bytes that never occur in the compressed data placed at pre-specified locations. It is a simpler approach in general, but less effec-

[9]Ternary digit is a base-3 number system that uses digits {0, 1, 2} only. In information theory, a ternary digit contains $\log_2 3$ information.

tive in HDFS and MapReduce. Recall that MapReduce is intended to process a record at a time. Inserting synchronization points on a per record basis will firstly degrade compression ratio. Secondly, synchronization points may differ depending on the data contents. If a MapReduce program needs to process a dataset that contains several separate files, it can be difficult to find common synchronization points (the same byte or group of bytes that is used as a synchronization point). If different synchronization points are used, it will make it more complex for the MapReduce program to determine records.

An alternative solution is to use self-synchronizing codes which can self-determine whether a codeword has been corrupted when it is received or within a synchronization cycle. Many self-synchronizing codes are suboptimal or only optimal under strict conditions [FT13] [Tit96] [Gol66] [FK96]. Temporarily putting code optimality aside, in order for Mapper to read records directly from, for example, Huffman compressed data, we need to determine record delimiters and check codeword integrity. At a large scale, this process can seriously impair overall performance of a MapReduce program. Also, note that synchronization points and self-synchronizing codes are not designed (due to context dependencies) to facilitate random access to data compressed by schemes using adaptive models.

### 2.3.3 Indexing

Indexing is one widely used technique that facilitates fast information retrieval and random access to data. Two common approaches to indexing are full-text indexing and self-indexing. Full-text indexing either builds indices on source data or compressed data. Inverted-file indexing [DHL92] [ZM06] builds an index on a text file by constructing a list of vectors. Each vector contains a distinct word/non-word (as a key) followed by a list of positions of the word/non-word occurring in the file. Searching on word(s) can be done quickly by looking up the index without referring to source data. In contrast, a Bitmap is a form of relaxed Inverted-file indexing [WMB99] [CWC$^+$15]. It does not store the exact position of words/non-words. Instead, a Bitmap only tells whether a word/non-word exists in a given source data.

A Signature File [Fal85] [ZMR98] builds indices using bit-vectors (also referred to as Descriptors or Signatures) and hash functions. For example, in order to create a Descriptor for text message "*text data*", each word in the message is hashed twice by two different hash functions. In Figure 2.8, in a simple form, words are hashed using the Fowler–Noll–Vo (FNV_1_1) [GFE12] and Murmur3 [10] functions. Each hashed value is modulated by the Descriptor length (16-bit in this example). The results are integer

[10]Murmur3 is a non-cryptographic hash function. It is commonly used by storage and database systems for fast information lookup. https://sites.google.com/site/murmurhash/

values indicating the positions projected on the bit-vector as shown in Figure 2.8 (*bottom*). The bit-vector is then used as the Signature/Descriptor of the message. Searching information from the Signature File requires hashing the pattern to be searched and using the bit-position values derived from the hash functions to verify whether the pattern exist in the source data. Note that there is a chance that two words can result in the same hash values, thus the Signature File only indicates whether a pattern exists in the source data with a certain level of confidence.

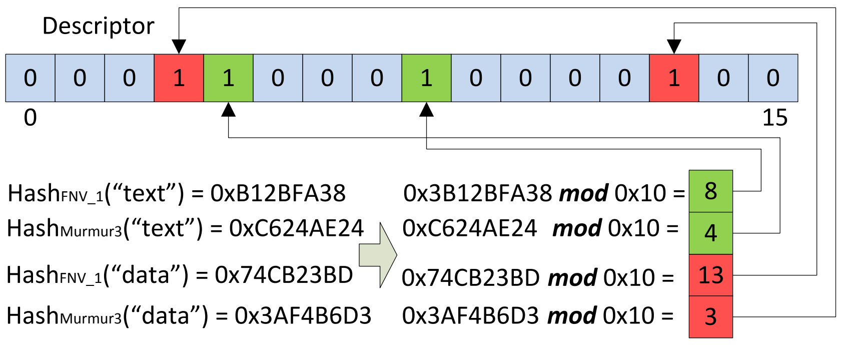


Figure 2.8: An example of Signature File indexing. Each message is hashed separately using FNV_1 (32-bit/x86) and Murmur3 (32-bit/x86).

Inverted-files, Signature Files and Bitmaps are techniques for building indices on source data. These techniques consume more storage space for faster information retrieval and so are suitable for database and storage systems. In this thesis, we aim to improve data analysis performance in Hadoop and MapReduce without sacrificing storage space.

Full-text indexing on compressed data is commonly based on *Rank* and *Select* operations [Jac88] [FLPnSP09]. A *Rank* operation calculates the number of occurrences of a symbol up to a given position in a source text. For example, given a text $T$(*text data*) with length 8 (ignoring the white space character), $Rank_x(T, 8)$ gives the number of occurrence of character "$x$" in $T$ which is one. *Select* is an "inverse operation" of *Rank*. $Select_x(T, j)$ returns the position of $j^{th}$ occurrence of character "$x$" in $T$. If given $j$ = 1, $Select_x(T, 1)$ will return three. *Rank* and *Select* are often used with *succinct* data structures [Jac88] to facilitate random access to compressed data. Figure 2.9 shows an example of how to use *Rank* and *Select* for random access to data compressed using a Wavelet Tree [GGV03]. The alphabet for $T$ is divided into left and right sets with symbols on the left indicated by "0" and right indicated by "1". Based on the left and right, we split the source data into two sub-strings. Each sub-string at the next level is treated as new independent source data and the splitting process continues until all leaves contain the same symbols as shown in Figure 2.9. As a result, symbols in the source data are rearranged such that the same symbols are grouped together. Similar to the BWT transformation, further compression algorithms such as Run-Length En-

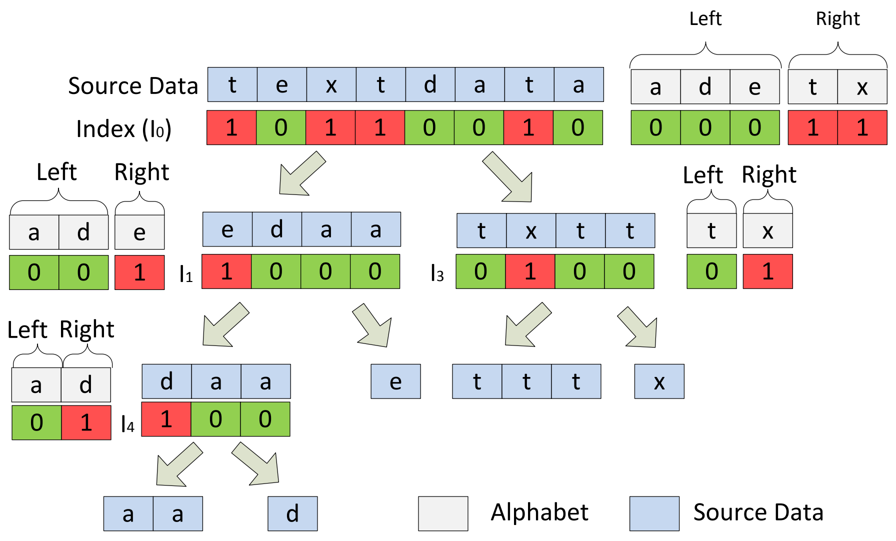


Figure 2.9: An example of using *Rank* and *Select* for random access in Wavelet Tree.

coding (RLE) [Sal08] can take advantage of the repetitiveness for compression. The binary bit sequences at each level of the tree are indices. The following shows how to count the number of occurrence of character "$a$" in $T$. Given that the alphabet $\Sigma = \{a, d, e, t, x\}$ for $T$ is known, dividing $\Sigma$ into two parts, we know that symbol "$a$" is at the left, therefore we need to calculate $Rank_0(I_0, \lceil \frac{N}{2} \rceil) = 4$ (returning the number of zeros in index $I_0$), where $N$ is the length of the source data. In the second step, we calculate $Rank_0(I_1, 4) = 3$ and calculate $Rank_0(I_4, 3) = 2$ in the third step. This gives the final results. A *Select* operation starts from the leaves and follows similar processes as demonstrated for *Rank*. In general, counting, locating and extracting arbitrary patterns can be achieved using a combination of *Rank* and *Select* operations.

*Rank* and *Select* operations can also be generalized to word level [FLPnSP09] or bit level [Jac92]. In [Jac92], a two-level index was built for Huffman compressed files. The index records the starting point of a codeword for every $k^{th}$ symbol. The scheme supports fast locating and extracting of patterns as codeword boundaries can be quickly identified by appropriate combination of *Rank* and *Select* operations. However, random access to compressed data requires having both the index and the compressed data. The authors [Jac92] indicate that the two-level index occupies the same amount of space as the compressed data. Thus, using bit-oriented Huffman coding with full-text indexing for random access will double the total space occupancy. This makes this scheme less attractive for big data.

In contrast, self-indexing is particularly interesting. It can be used standalone for counting, locating and extracting patterns from compressed data without decompression. Self-indexing builds indices for compressed data using *succinct* data structures.

FM-Index [FM00] and its variants [FMMN04] [GNS$^+$06] are typical self-indexing algorithms. FM-Index and Huffman FM-Index both use the Burrows-Wheeler Transformation (BWT) and *succinct* data structures such as suffix arrays for compression and indexing. But, this also poses problems for searching strings that span BWT blocks. Additionally, textual data processing often requires complex operations, for example, splitting a record by specified delimiters, merging multiple fields to form a new record, string to numerical value conversions. Using self-indexing thus makes an analysis program difficult to manipulate the underlying data as modifying data involves changing its corresponding indices or even requiring re-compressing of the data.

## 2.4 Emerging Techniques for Big Data Compression

Considering the dynamic nature of the compressed contents, block-based compression will work favourably for HDFS, because each compressed block can be made self-contained. Due to the variable size of the compressed blocks, an additional indexing process must be applied to the compressed contents to log the block sizes, so that the MapReduce *Record Reader* component can effectively and safely parse records. In principle, any block-based compression can be indexed. In practice, *LZO*-splittable [Twi] used in Twitter® implements such an approach. *LZO*-splittable uses standard *LZO* for data compression, then uses a separate program to scan through the *LZO* compressed data and log the block boundaries in separate files. In order to consume the *LZO*-splittable compressed data, a MapReduce program will need both the compressed data and the log file(s). If a dataset contains several files, each compressed file must be associated with a dedicated log file. This makes a MapReduce program more complex and error-prone due to the tracking of block boundaries.

As well as of *LZO*-splittable, [LRW11] builds inverted file indexing (Section 2.3.3) on compression blocks. This allows a MapReduce program to quickly identify desired information (records) in a compressed block without decompression. On the other hand, the *LZO* compression scheme is speed-oriented. It compresses data at a relatively low ratio. Adding extra indices to the compressed data makes the aggregate compression ratio even lower. More importantly, as Mappers are independent processes, partitioning and distributing indices with aware of data locality can be very problematic. Hadoop++ [DQRJ$^+$10] is other independent work on indexing big data for MapReduce. Hadoop++ does not relax the HDFS storage burden as indices are build on top of source data. Moreover, indices are built when source data is being uploaded to HDFS. This implies that the indices on source data are static. Different type of analysis jobs may not suit the same indices at all. In contrast, [RQRSD14] provides an adaptive indexing technique for HDFS. Indices are built gradually during the Map processes. The main drawback is that sharing the adaptively built indices across clusters may cause synchronization problems. [ZIP14] [YP09] [ZAW+09] are several other indexing techniques that have many common properties to the approaches discussed above. The majority of those developed for Hadoop and MapReduce are targeted on source data. They provide better performance for specific types of MapReduce jobs at a cost of more storage space and higher complexity.

Apache Avro [Apaa] is a data serialization system. It provides functionalities for organizing and splitting compressed data in specific Object Container File format. Each file consists of a schema (meta-data that describes the data, e.g., data size, block number and compression codec) and a series of data blocks. Each data block is self-contained and contains a group of compressed objects (records) and 16-byte synchronization markers which are used for determining boundaries between data blocks, thus making compressed data splittable. The inconvenience of using Avro is that:

- it requires maintaining a separate system for the organization of compressed data,
- transferring Avro compressed data to other systems (e.g., another Hadoop environment) requires the Avro system to be present and to be a compatible version.

In big data query systems, column-wise compression [FPST11] [MGL+11] [LAC+11] [LR13b] [HCG+14] [LOOW14] is commonly seen. [FPST11] uses a method that simply compresses column data into a series of self-contained blocks. A group of in-order blocks are then packed into HDFS-Blocks. This makes the compressed data splittable. However, coordinating column-wise blocks horizontally (columns aligned in rows) during data processing can be very difficult. Another scheme, Llama [LAC+11] allows grouped-column compression with different schemes best suitable for groups. It builds indices for compressed blocks. Hive [HCG+14] is a data management system that allows column-wise compression. Data is organized like database tables in a Optimized Record Columnar (ORC) format which is specific to Hive. Internally to ORC, data is partitioned into groups of records, namely *stripes*. Within each *stripe*, records are separated into columns. Each column is compressed twice. The first level compression is based on a dictionary method. This is because data in the same column is considered to have similar attributes and therefore shares a common vocabulary. Using a dictionary method can achieve better compression. The first level compression results are then forwarded to the second level compression which uses general purpose compressors such as *LZO* to pack data into fixed-size blocks (256KB). Hive ORC can potentially achieve high compression ratios and it is splittable. A concern is that ORC is data agnostic. Many datasets cannot simply be formatted in columns. For example, eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) formatted data often contains nested records. *Pzip* [BCC+00] (and its improvement [BFG03]) is another specialized compression scheme designed for table data. The algorithm uses heuristics to group column data so that the combination can yield better compression. However, *Pzip* relies on a strong assumption that the width of columns is fixed.

From another perspective, it has shown that I/O operations including reading data to Mappers, materializing data to HDFS and transferring intermediate data over the network, are the dominant factors that affect the overall performance of MapReduce [PPR$^+$09] [JOSW10]. To minimize network traffic, [ZYTC14] proposed an adaptive compression that compresses data according to network traffic patterns. This is more beneficial in cloud and HPC (High Performance Computing) environments where data traffic varies greatly in time. Hadoop is said to be *moving computation to data*. The network traffic is dominated by the MapReduce *Shuffling* process which is generally predictable. Hadoop also supports compression before delivering data to designated computational nodes.

### 2.4.1 Probabilistic Data Structures

Recently, we have witnessed the use of probabilistic data structures and associated algorithms for dealing with big data in computational biology [RH15] [SKA$^+$10]. Probabilistic data structures are based on Bloom Filters [11]. They provide an efficient way of verifying whether a given message exists in a dataset. Using Bloom Filters can significantly reduce data size as a group of messages will be transformed into a single finite series of bits [Blo70]. However, the information transformation is a one-way process. In other words, we can query a Bloom Filter, but we cannot retrieve information that has already been stored in it due to the use of hash functions [12]. This limits the applicability of probabilistic data structures to domain-specific use only, such as genome sequencing. Another application of probabilistic data structures is big data queries, for instance, BWand [AL13] for fast query on Twitter *tweets*, content filtering in MapReduce programs [MS12] and NoSQL (Not Only Structured Query Language) databases such as Google BigTable [CDG$^+$08], Apache HBase [Apae] and Apache Cassandra [Apab]. In these cases, the probabilistic data structures are used as an indexing technique to quickly locate information in a distributed storage system. Yet, the original datasets are still needed. There are many other constraints such as immutability and uncertainty that limit the scope of using probabilistic data structures in big data analysis, for example, when modifying the original messages or reading financial records. Generally, probabilistic data structures are suitable for assisting queries on large datasets and applications that can tolerate errors. They are in principle the same as Signature Files as discussed in Section 2.3.3.

[11]The Bloom Filter [Blo70] was first introduced by Burton Howard Bloom in 1970. Basically, a Bloom Filter is a $m$ bits array. A number $n$ original messages are fed into a number $k$ hash functions. Each hash function will produce a uniform randomly distributed bit sequence that maps to the Bloom Filter. It tells how likely a given message exists in the Bloom Filter given in probabilities defined by $(1 - e^{-kn/m})^k$.

[12]Note that most hash functions are one-way. Meaning that transforming a source message to a hash value is computationally easy whereas the reverse process is extremely hard.

### 2.4.2 Data De-Duplication

Data de-duplication is not a new idea. It derives from delta coding [KMMV02]. Traditionally, it is used in storage systems for backup services and software patching. The basic idea of data de-duplication is to organize information in a hierarchical structure in which the commonality of information flows from top to bottom. It is recently becoming popular for big data due to achieving very high compression ratios. Industry applications such as Google Drive [Goo], Dropbox [Dro] and RainStor [Rai] heavily rely on data de-duplication for saving storage space. RainStor has demonstrated that using composite de-duplication schemes can reduce data size by up to 97%. Although, the achievement on compression ratio is surprisingly good, retrieving original text can be time consuming due to the need to go through several de-duplication processes from byte-level to field-level. Interestingly, but not directly related, industrial implementations often use de-duplication across users (at service provider level) [ABKY15]. Special encryption techniques and mechanisms for verifying ownership of duplications [RMW15] [DAB$^+$02] are used for data protection. More specifically, [DKS08] and [KLA$^+$10] examined de-duplication for Uniform Resource Locator (URLs). The basic process is to extract URL patterns and define rules so that URLs can be fitted into pre-defined templates and repetitiveness thereafter can be eliminated. Generally, the pattern extraction and template-based de-duplication are very useful. But this approach is less generic. Depending on the data format, it is relatively easy to find patterns in URLs, but in other situations such as plain text, it is not obvious how to find patterns. Several similar schemes for de-duplication are summarized in [PP14].

### 2.4.3 Domain-specific Compression

Apart from general approaches, several data-specific compression schemes are also developed for accelerating domain-specific analysis. The ever-increasing popularity of using mobile devices produces large volumes of sensor data. Processing, storing and managing such discrete-time streaming data is challenging. In [FSR12], the authors propose an algorithm for approximating Global Positioning System (GPS) and Light Detection And Ranging (LiDAR) data. The compression is achieved using Line Simplification [AK93] [CWT06] with $k$ segments. Source data is segmented in such a way that data points in the range of the segment have minimum projection distance to the segment line. TribleBit [YLW$^+$13] is designed for storing Resource Description Framework (RDF [13]) data in a compact format while providing better resource query performance. TribleBit takes advantage of the fact that RFD data only contains three attributes for

[13]RDF is a specification defined by W3C (World Wide Web Consortium) for interchange of data on the Web. It is also known as a data modeling framework for the semantic Web. A RDF statement for a Web resource is comprised of a *Subject* (resource), *Predicate* (relationship between a subject and an object), and an *Object* [GB14].

describing a Web resource. Compression is achieved by compressing columns separately using a dictionary-method based on symbol-substitution. Indexes are built on top of compressed data. Because a dictionary-method symbol-substitution method is a context-free scheme, hence queries can be performed directly on the indices. [UMB10] proposed a parallel compression algorithm implemented in MapReduce for semantic Web data based on a word-substitution method. The authors in [UMB10] use a static dictionary as a compression model which can be replicated and distributed to Mappers, thus the compression of data blocks can be performed independently in parallel and compression results can be aggregated at Reducers for the final outputs. However, how to split the compressed data in HDFS and decompress HDFS-Blocks in parallel have not been addressed.

In [BbXb15], the authors propose a lossy compression scheme for big time-series data based on estimation methods with irregular sampling [bJfLSW13]. The scheme uses an adaptive mathematical model to capture features and regularities with corresponding sampling density. Features are sampled with more data points whereas steady states are sampled less, the total number of sampling points is generally less than sampling with fixed intervals, thus achieving compression. However, capturing features in a time-series is computationally expensive and defining what a feature is in a time-series depending on the type of analysis. [BO14] and [BOY14] are two lossy compression schemes designed for 3-Dimensional (3D) time varying data. The algorithms start by segmenting 3D data into time frames. Data points in a frame are converted into a point vector. This in fact transforms 3D data into a point matrix. Based on the point matrix, a base vector, a coefficient vector and a mean vector are calculated [Cha00]. Source data can then be approximated using the three vectors, thus achieving compression.

Considering the popularity and growth of streaming data (potentially from multiple sources), entirely software implementations of data-specific compression schemes may be inadequate for real-time compression. [JFAE12] introduced a hardware assisted scheme for online data compression using Field-Programmable Gate Arrays (FPGAs). It has been shown that the FPGA-based compressor can perform several times faster than general purpose compressors implemented solely in software. But the speed efficiency is offset by the complexity of porting compression schemes implemented in common programming languages to FPGA enabled operating systems such as LEAP [PAF$^+$10]. SIMD-Group-Scheme [ZZSY13] is another hardware specific compression for fast encoding and decoding of textual big data (online and offline) in parallel using Intel® SSE [14] instruction sets.

[14]SSE stands for Streaming SIMD (Single Instruction Multiple Data) Extensions. It is a CPU instruction set that is specific to video/audio/image encoding/decoding and string processing. SSE is available on both Intel® and AMD® CPUs.

## 2.5 Summary

Processing compressed data in parallel in distributed environments requires the compressed data to be splittable; a context-free scheme is an appropriate method that allows compressed data to be directly processable; being content-aware allows the seamless integration of compression, HDFS and MapReduce.

We have seen how big data is organized on the Hadoop platform. In order to consume compressed data transparently in a distributed environment, an effective compression algorithm must fulfil the requirements of being splittable and allow random access without sacrificing too much compressibility. Conventional compression schemes strive to achieve higher compression ratios by leveraging preprocessing and contextual information. As this process creates strong dependencies, data accessibility is forced to be sequential. In order to explore the possibility of accessing compressed data in parallel in distributed environments, we decompose conventional algorithms into three phases including *transformation*, *modeling* and *encoding*. We have identified that using *transformation* can make random access to compressed data extremely hard and incurs a high cost for compression/decompression due to permutation of symbols as demonstrated in Section 2.2.1. Avoiding *transformation*, an encoding scheme based on static or semi-static modeling techniques can often result in the desired properties allowing splitting and manipulating for compressed data. However, attention must be paid when using context-free schemes for big data processing and analysis.

It should be noted that Unicode is the dominant encoding scheme on the Internet at present. The majority of textual contents are encoded in, for example, UTF-8 or UTF-16 (Unicode Transformation Format 8-Bit/16-Bit) format. Current context-free schemes do not deal with Unicode data. More importantly, these schemes compress data without concern for the organization and format of the underlying data. This makes the compressed data non-transparent to the existing analysis algorithms and software packages. Therefore, new compression schemes, that are content-aware and able to effectively deal with Unicode contents, are needed.

A drawback of using static or semi-static models is relatively poor compression. In contrast, employing adaptive modeling techniques (higher-order compression) generally yields much higher compression ratios, but creates strong dependencies. Special indexing techniques have been developed to allow random access to compressed data.

Indexing is an effective mechanism for fast information retrieval from large datasets. Full-text indexing builds indices on top of source data or data compressed using context-free schemes. This makes the scheme less attractive for big data. In contrast, self-indexing can build indices for data compressed by higher-order entropy schemes. It provides better compression ratios while allowing several primitive operations on data without decompression. However, self-indexing presents shortcomings on data structure complexity and ease of data splitting. In order to consume data, in a distributed environment, that has been compressed by a higher-order compressor, and avoid using these complex indexing techniques, a new compression format is needed.

Emerging techniques developed for big data mainly focus on reducing data size for saving storage space. In general, these techniques are data- or application-specific schemes which often result in much higher compression ratios as the algorithms are optimized for the particular data with respect to format and contents, compared to general purpose compressors. Considering the variety of big data, maintaining heterogeneous data requires managing multiple data- or application-specific compression schemes. This makes the current emerging compression schemes specific to datasets and uni-functional.

Overall, we see that conventional compression schemes are not designed for use in distributed environments. Existing algorithms seem to deal poorly with the new characteristics brought by big data. The emerging techniques are less general and similar work such as *LZO*-splittable still need to be improved. Thus, new compression schemes are needed. In the following chapters, we introduce content-aware compression schemes that are designed for large scale textual data while allowing the compressed data to be consumed in parallel in the distributed analytical environment of Hadoop.

# Chapter 3

# Content-aware Partial Compression (CaPC)

*"Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains."*

— Steve Jobs

Previously, we have discussed several existing methods for data compression. The advanced implementations of those compression schemes are very effective in terms of compression ratio. These algorithms generally reduce data size by using context information and disregarding the organization of data contents, for example, information ordering and data format. This makes accessing the compressed data extremely difficult without decompression. There are several special cases where compressed data can be searched, but at a cost of high complexity [Ven14]. In contrast, a context-free compression scheme (also referred to as context-independent compression), in principle, allows direct data manipulation without decompression.

In this chapter, we introduce the Content-aware Partial Compression scheme (CaPC) [DH14a] for textual big data analysis on Hadoop. We take a naïve dictionary-based approach by simply replacing any matched string by its corresponding code, while maintaining the original properties of the data, such as information ordering, data format and punctuation. The rationale behind CaPC is that any meaningful string, for example, an English word, is only meaningful to humans. A string and its transformed form (code) do not make any difference to machines, as long as the machines process them in the same way. In contrast, functional characters are usually meaningless to humans but functionally useful, for example, a white-space character is commonly used as a delimiter to separate words. Additionally, information is often organized in a certain format, for example, JSON or XML, so that the data can be easily understood by a suitable interpretor, parser or algorithm. If we replace strings with shorter codes while keeping the functional information intact, then the compressed data can be made transparent to analysis algorithms.

In addition, CaPC provides a unique feature that the codes are valid strings, and they are compatible with ASCII and UTF-8 encoding schemes, rather than simple binary sequences. This is particularly useful for the MapReduce computational framework. The current version of MapReduce only supports ASCII and UTF-8 [1] schemes, and a MapReduce program will automatically convert text data to UTF-8 strings during the data loading process. Therefore, CaPC is transparent to MapReduce. As part of CaPC, a set of supporting utility functions are also provided to assist the development of CaPC enabled MapReduce programs. These functions transform strings or string variables into their corresponding codes and deal with regular expressions at the program compilation stage.

CaPC can be considered as a lossless compression scheme under the assumption that strings are case-insensitive. This condition holds for many analyses, for example, sentiment analysis, semantic analysis and PageRank calculations. We evaluate CaPC using a set of real-world datasets with several standard MapReduce jobs on Hadoop. The evaluation results show that CaPC works well for a wide variety of analysis jobs. CaPC can achieve $\sim$30% data size reduction on average and up to $\sim$32% performance increase for I/O intensive jobs. While the gains may seem modest, the point is that these gains are "*for free*" and act as supplementary to all other optimizations. Moreover, CaPC consists of two independent parts. The first part is the compression algorithm. It is implemented in `C++` for better performance. The second part is a utility library implemented in Java for MapReduce programs. Thus, CaPC is provided as a integrated solution in MapReduce rather than a standalone compression scheme.

## 3.1 Compression Scheme

The CaPC architecture is shown in Figure 3.1. CaPC takes source data and a static *base-dictionary* as inputs and produces compressed data and two compression model files. The first model file is the static *base-dictionary*. The second model file is built when the compression process starts, as detailed in Section 3.1.1. The compression process is as simple as replacing any compressible string found in the compression models by its corresponding code. The *base-dictionary* is essentially a static word-list which contains $\sim$60 thousand of the most frequently used English words (spoken and written)[LRW01]. The length of each word (number of characters in the word) in the *base-dictionary* must be greater than four. This is limited by our coding scheme as detailed in Section 3.1.2. Before starting an analysis, the CaPC compressed file(s) and

[1]MapReduce uses UTF-8 encoding scheme for strings by default.

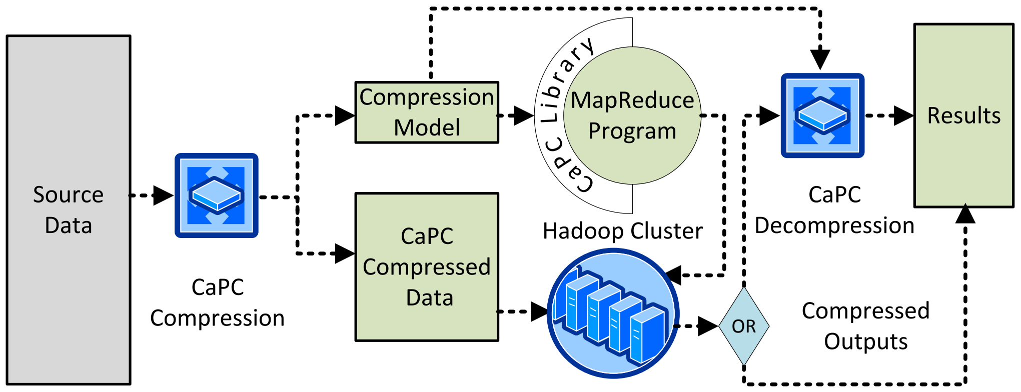
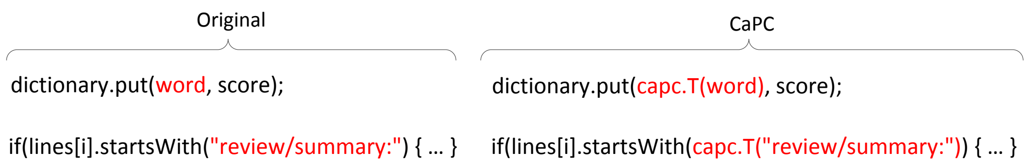


Figure 3.1: Content-aware Partial Compression architecture overview


Original
CaPC

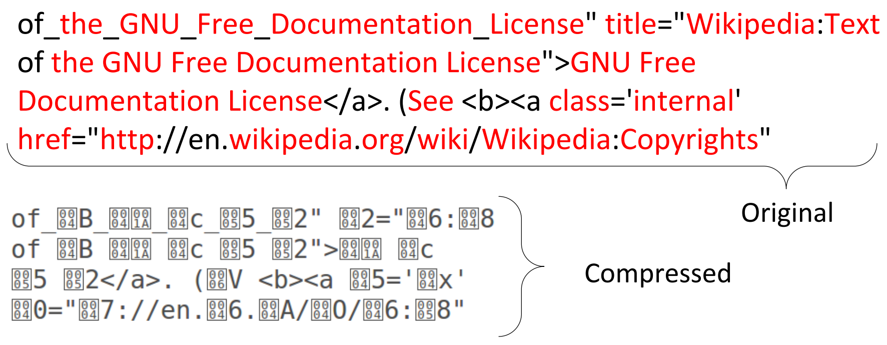
```
dictionary.put(word, score);                           dictionary.put(capc.T(word), score);

if(lines[i].startsWith("review/summary:") { ... }      if(lines[i].startsWith(capc.T("review/summary:")) { ... }
```

Figure 3.2: Examples of using CaPC in a MapReduce program.

its associated model files must be loaded into HDFS and HDFS Distributed Cache [2], respectively. In a MapReduce program (existing or new), any string or string variable needs to be enclosed by the CaPC library function - *T( )*, as demonstrated in Figure 3.2. The *T( )* function transforms a string or a string variable into its corresponding code during the *Mapper* and *Reducer* initialization and processing phases. This guarantees that the MapReduce program can work with CaPC compressed data directly without decompression. Other utility functions are available to deal with aspects such as regular expressions. Finally, analysis results are in CaPC compressed format. They can be optionally decompressed to store directly in HDFS.

In designing CaPC, preserving the original data format is one of the primary goals. To comply with this, any functional character must be untouched, and the characters used for codes must not contain any functional characters. This is mainly driven by the fact that existing algorithms or parsers rely on the data format and/or pre-defined functional characters for processing, for example, XML or JSON parsers. A string that

[2]HDFS Distributed Cache is a temporary storage space drawn from the Hadoop distributed file system and provided for MapReduce programs to cache application-specific files, for example, $3^{rd}$ party libraries and text files. Transmitting files from HDFS Distributed Cache to Hadoop computational nodes generally through the network. It has implications of transmission delays.



```
of_the_GNU_Free_Documentation_License" title="Wikipedia:Text
of the GNU Free Documentation License">GNU Free
Documentation License</a>. (See <b><a class='internal'
href="http://en.wikipedia.org/wiki/Wikipedia:Copyrights"
```

```
of_␄B_␄␚_␄c_␅5_␅2" ␄2="␄6:␄8
of ␄B ␄␚ ␄c ␅5 ␅2">␄␚ ␄c
␅5 ␅2</a>. (␆V <b><a ␄5='␄x'
␄0="␄7://en.␄6.␄A/␄0/␄6:␅8"
```

Figure 3.3: An example of CaPC compressed data comparing to the original data.

can be compressed must meet two conditions:

- It must contain consecutive characters in [a - z] and [A - Z] inclusively.
- The length of the string must be longer than the length of its corresponding code (2 or 4 bytes).

The compression process breaks a text file into string tokens separated by functional character(s). In CaPC, any character outside of the range [a - z] and [A - Z] is considered as a functional character. Each string token will be searched for in the compression models. If it is found, the string token will be replaced by its corresponding code. Otherwise, it will be sent to the output unchanged. Figure 3.3 shows an example of a CaPC compressed file in comparison to its original format. Note that, the first condition given above means that numerical information is not compressible. This is due to the fact that if the numerical string is a data value (for example, in a financial report), it will be infeasible to generate a unique code for each value; if it is used as text (for example, part of a user ID or product ID), extra information about the context is needed in order to understand how the numerical string should be interpreted. To be more generic, CaPC does not compress numerical information. In the following sections, we introduce the codeword design and model construction strategies used for CaPC.

### 3.1.1 Compression Model

The basic technique used in CaPC is to replace any compressible string by a shorter code. As mentioned in Section 3.1, the *base-dictionary* contains approximately 60 thousand commonly used words. In fact, for a given text data, the majority of the contents may not be regular English words. For example, the *Memetracker memes* dataset [LBK09] mainly contains a big list of Website URLs, and the most repetitive strings are "*http*" and "*https*" which are not found in our *base-dictionary*. As another example, Twitter tweets are usually organized in JSON format when collected through

Twitter Stream APIs. The Twitter data contains a considerable amount of repetitive strings, for example, JSON attribute names, some of which are not in a regular form, for example "*id_str*". To deal with application-specific strings and improve the compression ratio, the CaPC compression model consists of two dictionaries: *sampled-dictionary* and *base-dictionary*. The *sampled-dictionary* is much smaller in size. It contains the most frequently used compressible strings gathered from a sampling of the input data. It takes priority over the *base-dictionary*. In another words, our search algorithm will firstly examine the *sampled-dictionary* since there is a higher probability a match will be found there. If there is a hit, the string will be replaced by its corresponding code, otherwise, the *base-dictionary* is searched.

Prior to the actual compression process, CaPC takes samples from each file of a given dataset. From the samples, we start building a word frequency table by simply counting the occurrence of each word that is observed. After the sampling process is complete, the frequency table will be sorted by word count and words of the same frequency will be sorted by word length. The top 888 words (this limit is due to our coding scheme as detailed in Section 3.1.2) will be selected for the *sampled-dictionary*. Note that the sampling process is on a per dataset basis. If a dataset contains multiple files, there will be only one *sampled-dictionary* built for the dataset. For accumulated data, if the initial dataset is large enough and the data comes from the same source, for example, Twitter tweets or server log files from the same data centre, the compression model generated from the initial dataset can be directly used by CaPC for compressing incremental data without a sampling process. This can dramatically improve compression performance. A similar strategy can also be used to compress very large datasets where a small portion of the data can be selected to do an initial compression. The rest of the data will be compressed using the generated compression model(s) without sampling. Detailed sampling strategies are given in Section 3.1.3.

Note that there is no overlap between the *sampled-dictionary* and the *base-dictionary*. The maximum size of each dictionary is limited by our coding scheme.

### 3.1.2 Coding Scheme

Codes generated for the *sampled-dictionary* have fixed length of two bytes and for the *base-dictionary* have fixed length of four bytes. Each code starts with a signature character followed by one or three characters. We choose a number of *control characters* [3] from the ASCII code table (Appendix D) as signature characters. 12 control characters are used for the *sampled-dictionary* and one is used for the *base-dictionary*. Referring to the CaPC compression example in Figure 3.3, the special character ␄ (the $4^{th}$ charac-

[3]Control characters are non-printing characters that do not represent written symbols. They are used for device signalling. For example, the *Carriage Return* character (*0x0D*) triggers a printer to start a new line.

ter) is a signature character indicating the beginning of a compressed symbol [4]. A full list of signature characters used in CaPC is given in Table 3.1. The reason for choosing these ASCII codes is due to the fact that Hadoop currently only supports standard ASCII and UTF-8 encoded text and these codes are valid characters for both standard ASCII and UTF-8 encoding schemes, but they are never used for textual data.

Table 3.1: A table of signature characters used in CaPC compression scheme.

| Hex | Char | Hex | Char | Hex | Char | Hex | Char |
|---|---|---|---|---|---|---|---|
| $0x04^*$ | EOT | $0x05^*$ | ENQ | $0x06^*$ | ACK | $0x0E^*$ | SO |
| $0x0F^*$ | SI | $0x12^*$ | DC2 | $0x13^*$ | DC3 | $0x14^*$ | DC4 |
| $0x15^*$ | NAK | $0x16^*$ | SYN | $0x19^*$ | EM | $0x1A^*$ | SUB |
| $0x11^+$ | DC1 | | | | | | |

*: used for *sampled-dictionary*; +: used for *base-dictionary*.

Recall that CaPC needs to preserve any functional character that appears in a given text. Thus, the available characters for codes are limited and they are taken from [0 - 9], [a - z], and [A - Z] inclusively. Signature characters are also valid for codes (acting as part of the code rather than a signature character). Due to this constraint, the *sampled-dictionary* can encode 888 words, because codes for the *sampled-dictionary* are two bytes long. The first byte is the signature character which is one of the 12 characters listed in Table 3.1. The second byte can be any one of the 12 signature characters or comes from the range listed above. The total code space is therefore given by 12*(10 * Number Symbols + 26 * Upper Case Letters + 26 * Lower Case Letters + 12 * Signature Characters) = 888. The *base-dictionary* has a total code space of $\sim$250 thousand which is given by (1 * Signature Character) * (10 * Numbers Symbols + 26 * Upper Case Letters + 26 * Lower Case Letters + 1 * Signature Characters)$^3$ = 250047.

Because CaPC code characters are drawn from the range [0 - 9], [a - z] and [A - Z] inclusively, and signature characters are valid for both ASCII and UTF-8 encoding schemes, the compressed data is also a valid text file. This makes the CaPC compressed data freely available for processing and analysis. In principle, any valid character can be used as a code, for example the *full stop* ($\cdot$) or the *question mark* (?) characters. There are mainly two reasons why we cannot use them:

1. for preserving the transparency between compressed data, the existing algorithms and software packages.

   For example, given a code "$0x11A \cdot A$" which is comprised of a signature character "$0x11$" and two characters "$A$" and a *full stop* character "$\cdot$". If the *full stop* character

[4]A symbol can be a word that is separated by specified delimiter(s) or a single character. It depends on the context. For example, in the CaPC compression, a symbol implies a word. In other cases, for example in the AHC (Chapter 4) compression, a symbol is a character.

"·" is used as a delimiter in the given text, then using a blind searching algorithm, the code will be separated into two parts: "$0x11A$" and "$A$" which will make the first part undecodable and the second part another string. In order to avoid this situation, we cannot use these functional characters.

2. to avoid broken codes in distributed file systems.

   Recall that HDFS organizes big files by splitting them into a series of blocks and then the data blocks are distributed across cluster nodes (*DataNodes* in the context of Hadoop). The data splitting process is done simply by size. Each data block will be assigned to a dedicated Map process. Moreover, at the boundary of each data block, the MapReduce *Record Reader* will check the logical completeness of a record. If it contains a partial record, the remaining part of the record will be streamed to the local (current) Mapper from the immediate succeeding HDFS data block, which is very likely located on anther physical *DataNode*. Using the same code as above, if the data splitting occurs after the *full stop* character, the string "$0x11A\cdot$" will be included at the end of the first data block, and the "$A$" will be at the beginning of the second data block. Because the full stop character is at the very end of the first data block, the *Record Reader* will assume this data block does not contain a partial record (assuming the *full stop* character is used as the delimiter). Thereafter, the string "$0x11A\cdot$" and "$A$" will be processed by completely independent Map processes and eventually produce incorrect results.

Since the code space is limited, we need to use the codes efficiently in order to achieve as high a compression ratio as possible. We use sampling techniques to gather statistical information about the data, so that shorter codes can be assigned to more frequently occurring words.

### 3.1.3 Sampling Strategy

The main objective of the sampling process is to build an approximate word probability distribution table, so that we can get an insight in a timely fashion into which words are most frequently used. Designing an appropriate sampling procedure is crucial to get relatively accurate estimates, therefore resulting in better compression. Research results from computational linguistics show that biased sampling is a well known issue that leads to an inaccurate estimate of vocabulary size [Tuk77]. In parallel, a study from lexical statistics indicates that data length can heavily influence both vocabulary richness [5] and word probability distribution [TB98]. These studies suggest that samples taken should be evenly spread across the entire data. Empirical studies also demonstrate that textual cohesion is one of the main factors that brings a bias to estimation of vocabulary richness [Baa96]. In addition, the authors state that words do not ran-

[5]Vocabulary Richness describes the lexical diversity of a given text.

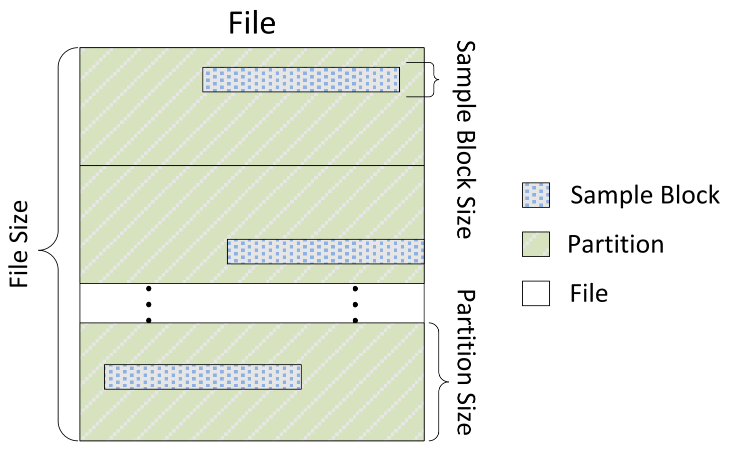


Figure 3.4: An illustration of CaPC data sampling strategy.

domly occur in a meaningful text. As a result, samples should be taken randomly and on a per-sentence basis. In practice, we use a block-based sampling strategy instead of a per-sentence approach for efficiency reasons. In summary, these previous studies led us to a sampling strategy of Stratified Random Sampling with Non-replacement.

Given an input dataset, each individual file will be treated as a stratum, hence stratified. Besides, each stratum will be partitioned and within each partition a randomly positioned block of data will be sampled, hence random. Partitions are non-overlapping, meaning that any piece of data will not be sampled twice, hence non-replacement. The number of partitions $P$ is simply determined by the sample size $n$ and sample-block size $b_s$, given by, $P = \lfloor n/b_s \rfloor$. This is illustrated in Figure 3.4. It serves as a general sampling strategy throughout this thesis. Thereafter, word frequencies are collected from samples. Note that although Complete Random Sampling with Non-replacement has better statistical properties, it is found to be inefficient, especially when the sampling rate is relatively high, due to the need to track sample positions to ensure non-overlapping samples. To comply with the research results from [TB98] and considering compression speed, the default sampling rate is chosen to be logarithmically proportional to the file size.

### 3.1.4 Algorithm Efficiency

CaPC compression space efficiency mainly depends on the quantity of compressible strings and effectiveness of the sampling process. The CaPC compression ratio $\varphi_{CaPC}$ for a given textual dataset can be calculated using Formula 3.1, where $n$ is the total number of characters in the dataset; $f_i$ and $f_j$ are the frequencies of word $i$ and word $j$ from the *sampled-dictionary* and *base-dictionary*, respectively, that have been observed in the given dataset; $l$ is the length of the word; *SD* and *BD* indicate the size of *sampled-dictionary* and *base-dictionary*, where *sampled-dictionary* $\bigcap$ *base-dictionary* $= \phi$.

$$\varphi_{CaPC} = \frac{\sum_{i=1}^{SD} f_i(l_i - 2) + \sum_{j=1}^{BD} f_j(l_j - 4)}{n} \tag{3.1}$$

Recall that CaPC does not compress numerical data and leaves the functional contents intact. The main processing in CaPC is searching string tokens in either the *sample-dictionary* or *base-dictionary* and replacing the matched strings with their corresponding codes. Because both the *sampled-dictionary* and *base-dictionary* are static (no insertion, modification or deletion during the compression), therefore, they can be organized using *HashMap* data structures [6]. This allows searching and retrieving a string from either the *sampled-dictionary* or *base-dictionary* in $O(1)$ time. Also considering that functional contents and numerical information are left intact, this leads to a sub-linear compression time for CaPC. The decompression process is the same as compression where finding the code (*key*) in the *HashMap* data structure (decompression model) and replacing it by the corresponding original string also takes sub-linear time.

## 3.2 Evaluation

We evaluate the effectiveness of CaPC using an on-site Hadoop cluster. The cluster consists of nine nodes, each with Dual core Intel E8400 (3.0GHz), 8GB RAM, and disk storage of 780GB 7200RPM SATA drives. All nodes are connected to a Cisco 3560G-24TS Gigabit Ethernet switch. All nodes run Linux kernel version 2.6.32 and Hadoop version 2.0.0. We use a set of real-world datasets for the experiments as listed in Appendix A (Table A.2) with various standard MapReduce jobs as summarized in Table 3.2. We deliberately choose these experiments to demonstrate various aspects of introducing CaPC to MapReduce, including analysis performance, cluster storage requirements and memory constraints over a wide range of problems. Specific to the underlying distributed file system (HDFS), it is configured with three replicas and a 128MB data block size. Snappy compression [Sna] is used for compressing the intermediate data across all experiments.

[6]A *HashMap* data structure is essentially an associative array. It is a structure that can map *keys* to *values*. It uses hash function to compute an index into an associative array from which the desired value can be found in $O(1)$ time. For example, if a given *HashMap* contains two pairs of key/value {{*big-data*, *code-1*}, {*small-data*, *code-2*}}, the keys "*big-data*" and "*small-data*" will be hashed separately and the hashing result is an integer value indicating the index of each corresponding key/value pair in the *HashMap*. In searching a pattern, for example, the string "*small-data*", we need to hash the pattern, then locate the key/value pair based on the hashing results.

Table 3.2: A summary of MapReduce jobs used for evaluating the CaPC compression scheme.

| Job | Dataset | Input Type | Input | Intermediate | Output | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|
| Sentiment Analysis | DS-Amazon* | O : | 9.6 GB | 23.3 GB | 7.1 MB | 5m52s | | |
| | | C : | 6.5 GB | 23.3 GB | 7.1 MB | 5m48s | 0.1% | 32.3% |
| PageRank | DS-Memes* | O : | 19.7 GB | 4.5 GB | 1.5 GB | 3m45s | | |
| | | C : | 14.5 GB | 3.7 GB | 1.1 GB | 3m43s | 0.1% | 26.4% |
| 5-Gram | DS-WikiEN* | O : | 40.0 GB | – | 144.8 GB | 17m8s | | |
| | | C : | 26.9 GB | – | 97.0 GB | 11m38s | 32.1% | 32.8% |
| WordCount | DS-WikiEN* | O : | 40.0 GB | 48.8 GB | 1.8 GB | 9m3s | | |
| | | C : | 26.9 GB | 35.7 GB | 1.3 GB | 8m0s | 11.6% | 32.8% |

O: Original datasets;
C: CaPC compressed datasets.
*1GB = 1073741824 Bytes*
∗: Subset of the data were used in the evaluation

### 3.2.1 Performance

In the *Sentiment Analysis* job, we evaluate how CaPC works with third party libraries. We use the *SentiWordNet* lexicons [ES06] to mine opinions from the *Amazon movie review* dataset [ML13]. The dataset contains ∼8 million reviews. For each movie, our MapReduce job takes each single word from the "*summary*" and "*review*" sections from all reviewers who have reviewed this movie, then checks against the entire vocabularies in *SentiWordNet* to get a positive or negative score. The final rating score is the sum of scores from all reviewers. The original lexicons in the *SentiWordNet* are in plain English. In order to work with CaPC compressed data, the lexicons need to be converted to CaPC codes. However, this additional process does not require any change to the existing MapReduce program logic. We only need to enclose variables or strings with the provided CaPC library function *T( )* (As shown in Figure 3.2).

Referring to Table 3.2.1, for this particular dataset, CaPC reduces data size by 32.3%. We observe no performance gains from the job using CaPC compressed data. This is due to the fact that the CaPC model files are initially uploaded to the HDFS Distributed Cache (referring to Figure 3.1) during the Mapper and Reducer initialization phase. The model files need to be loaded on each computational node from the HDFS Distributed Cache and then a string to code map must be built up by the CaPC library. This extra overhead, plus the fact that the analysis duration is short, neutralize the advantage of using the CaPC compressed data. With larger datasets, we can expect better performance gains. Also, notice that the Map outputs (intermediate data) and the Reducer outputs have the same size for both jobs using the source data and the CaPC compressed data. This is because CaPC does not compress numerical values and the results are lists of movie alphanumeric identifiers and their associated scores.

The *PageRank* experiment is chosen for demonstrating the advantage of using CaPC when analyzing with skewed records (where the length of the record is exceptionally large or small). Using the *Memetracker* dataset, the Map processes produce some highly skewed records that expand to approximately 800MB in size. This requires adjusting the *Java Heap Space* accordingly to accommodate such large records in memory. For the original source data, the MapReduce job requires the parameter *mapred.reduce.child.java.opts* = 1GB to successfully avoid the *Java Heap Space Error*. In contrast, 768MB is sufficient for the same analysis with the CaPC compressed data. Also, notice that there are no performance gains for this analysis. This is due to the same reasons given in the *Sentiment Analysis* discussion.

In contrast to computationally hard jobs (*Sentiment Analysis* and *PageRank*), we use *5-Gram* analysis to evaluate the impact of CaPC on disk I/O and network I/O intensive jobs. The jobs are optimized with 320 and 240 Mappers for source data and CaPC compressed data, respectively. The *5-Gram* job only has a Map phase, so the majority of the time is spent on loading the input data to memory and materializing the Map outputs

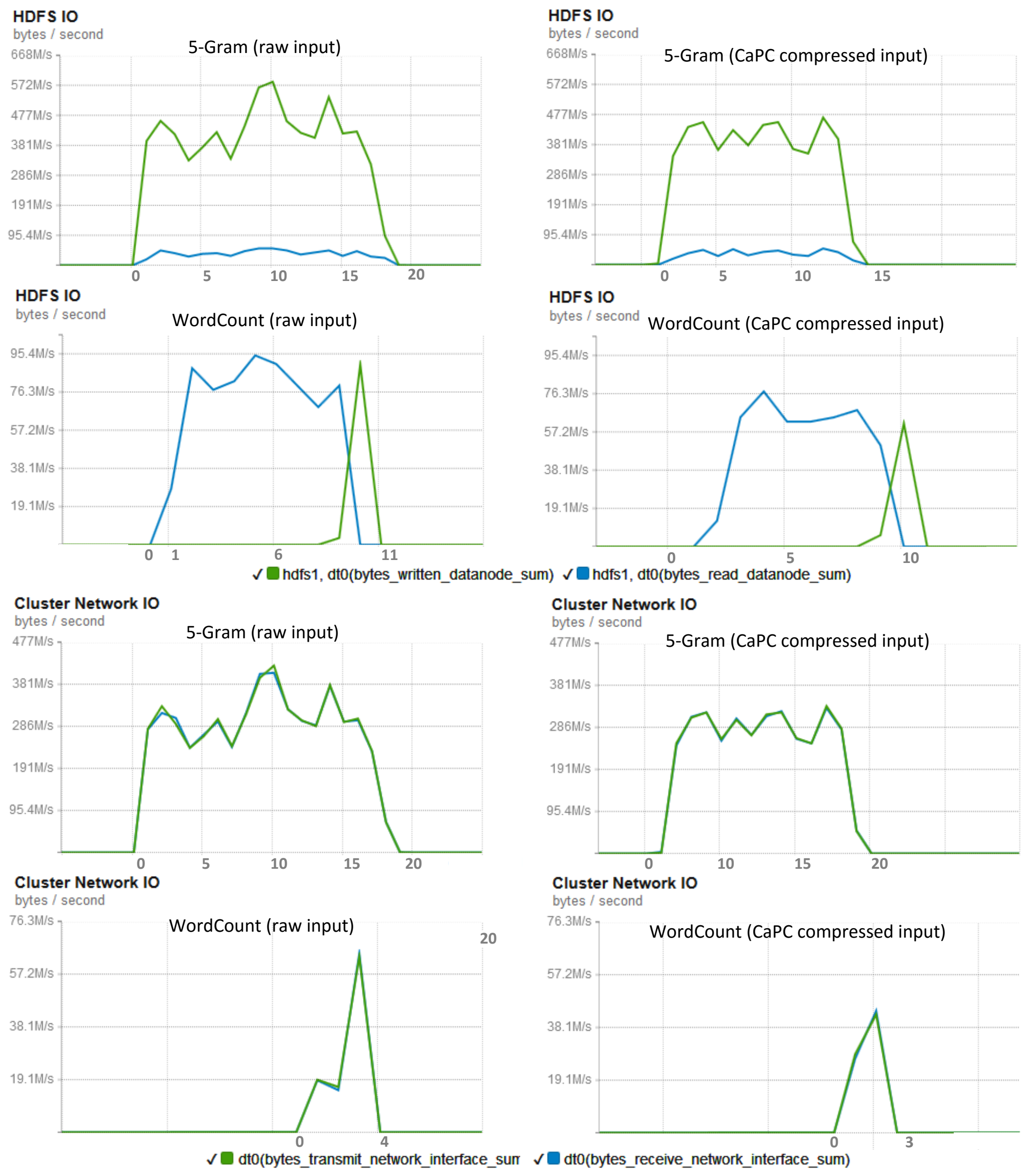


Figure 3.5: Statistics on HDFS I/O and network I/O for *5-Gram* and *WordCount* jobs with original data and CaPC compressed data, respectively. (X-axis indicates job durations)

to HDFS. In Figure 3.5 and Figure 3.6, we observe heavy disk writing operations on HDFS. Using the CaPC compressed data, we obtain a 32.1% speed-up compared to the same analysis with the source data. CaPC reduces the input and output data (still in CaPC compressed format) size by 32.8% and 33%, respectively.

In the *WordCount* job, we observe a 11.6% performance gain by using CaPC compressed data. Four sources contribute to the overall performance gains.

1. CaPC compressed data is 32.8% smaller than the original data size. Loading

smaller data from disks to memory takes less time.

2. The majority of the time spent on the *WordCount* job is finding word tokens. The process involves searching for delimiter(s) in the text and extracting string tokens individually. By default, using the *indexOf( )* function of the *String* class (provided by the Java library) for pattern searching requires $O(m(n - m))$ time, where $m$ is the length of the pattern string to be searched, $n$ indicates the length of the underlying text. Using CaPC compressed data, $n$ has been shortened by 32.8% on average for this particular dataset. The $m$ remains the same because in this case, pattern strings are delimiters and they are functional characters. CaPC does not compress functional contents. This leads to a theoretical time complexity of $O(m(0.672n - m))$ for the same pattern search with CaPC compressed data.

3. After the Map phase, the intermediate outputs need to be distributed to their corresponding Reducer nodes over the network. Using CaPC compressed data, the Map processes produce 26.8% smaller intermediate output data. Transmitting smaller data over the network takes less time.

4. Because the Reduce nodes also process CaPC compressed data, the final outputs are still in CaPC compressed format, which is 27.8% smaller than the outputs produced by the same analysis with the original data. Materializing smaller data to HDFS takes less time.

### 3.2.2 Further Compression

In order to efficiently distribute data between cluster nodes, Hadoop allows compression on intermediate data with options of several compression algorithms, including *Gzip* (*v1.4*), *Bzip* (*v1.0.6*), *LZO* (*v1.03*), and *Snappy* (*v1.1.0*). *Gzip* and *Bzip* algorithms offer better compression ratio; *LZO* and *Snappy* are speed-oriented. In CaPC, since data is encoded, the original patterns of the data contents have been changed. This will affect the compression ratio of the aforementioned algorithms. Taking *Bzip* as an example, it is one of the best compression algorithms and achieves a very high compression ratio by employing a block-wise Burrows-Wheeler Transformation (BWT) process. The BWT transformation results in an interesting result where similar characters tend to be grouped together, and hence repetitive patterns are identified for further stages of compression. In CaPC, given a fixed size of BWT block, it will fit in more CaPC compressed data, but the results may look more random compared to the results produced from the original text. If the CaPC compressed data contributes negatively to these compression algorithms, it will consequently harm the efficiency of data distribution over the network during the MapReduce *Shuffling* phase. In Figure 3.7, we use Hadoop supported compression algorithms to compress original data and the CaPC compressed data. We found that, although the CaPC encoding changes the original patterns of the

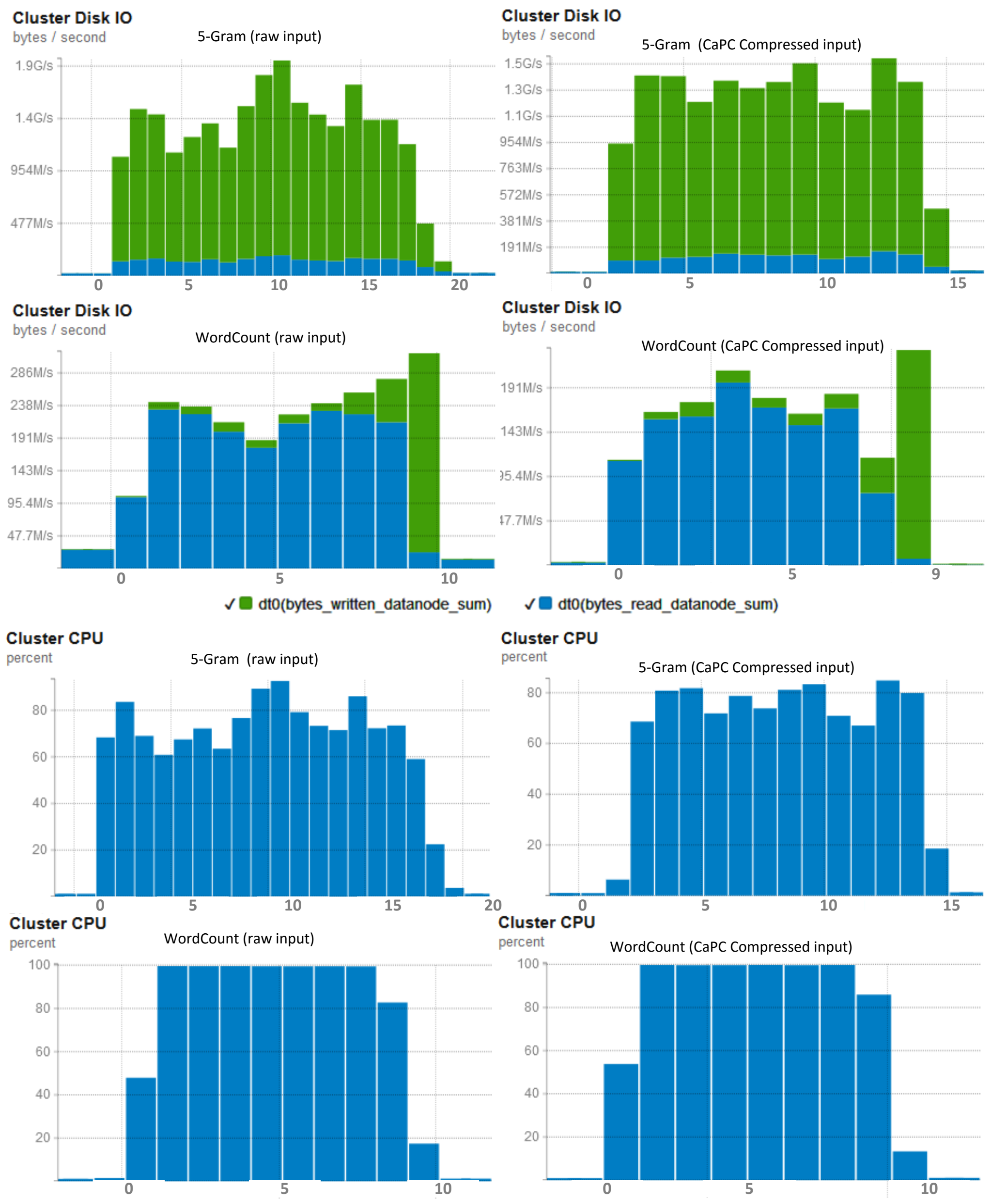


Figure 3.6: Statistics on Disk I/O and CPU utilization (*DataNodes*) for *5-Gram* and *WordCount* jobs with original data and CaPC compressed data, respectively. (X-axis indicates job durations)

data contents, it still results in smaller size. The results suggest that applying further compression when distributing intermediate data during the *Shuffling* phase can further improve the overall performance of analysis when using CaPC compressed data.

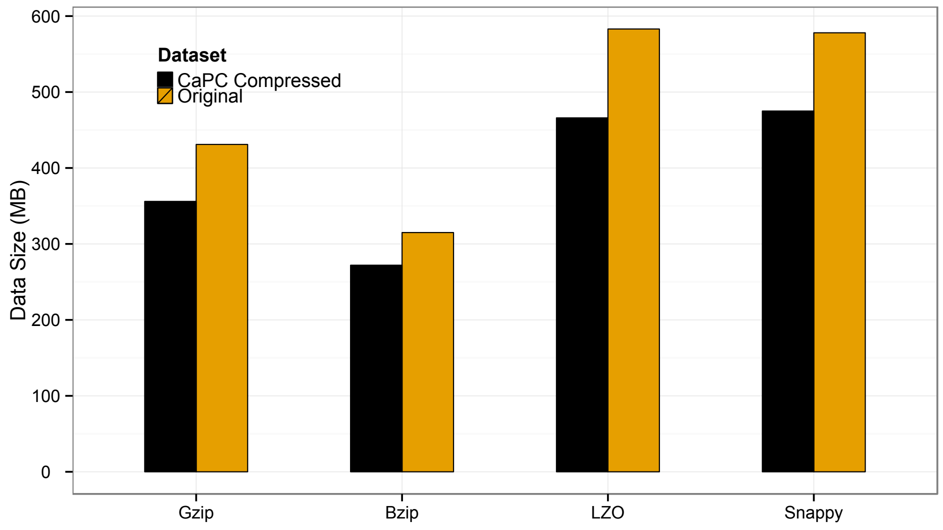


Figure 3.7: Applying compression on original data and CaPC compressed data. The original dataset is 1GB of Wikipedia articles in XML format and the CaPC compressed original data is 686MB.

### 3.2.3 Summary

Overall, we can reduce by approximately 30% the storage space required for storing data. This is not only for persistent storage. Performing analysis with CaPC compressed data can also reduce the volatile memory required, for example, the Java Heap Space, as demonstrated in the *PageRank* experiment. In the *Sentiment Analysis* and *PageRank* calculation jobs, we observe no performance gains. On the other hand, the *5-Gram* and *WordCount* analysis show a large difference in performance. Comparing the input, the intermediate and the output data size, as well as taking our cluster configuration into consideration, we conclude from these experiments that the performance gains largely come from the more efficient data distribution over the network and loading smaller data from persistent storage to memory. Hence, a Hadoop cluster which has limited network and storage will find CaPC beneficial, especially for data-centric analysis.

### 3.2.4 CaPC Characteristics

We use several state-of-the-art compression algorithms, including *Bzip*, *Gzip*, *LZO* and *LZMA*, as benchmarks to evaluate CaPC, although the purpose of CaPC is different from the others. Table 3.3 lists the compression/decompression speed, memory consumption and compression ratio for each algorithm. Because CaPC needs to preserve the original format of the data and uses a selective compression approach, it results in the least compression ratio. The performance hits for CaPC mainly come from the sampling process. In the experiment, the sampling rate was set to 0.1%. The sampling rate is adjustable. A higher sampling rate may offer a better compression ratio but poorer performance. Dictionary lookup is implemented using *HashMap* data structures which give *O(1)* time complexity. However, as mentioned in Section 3.1.4, *HashMap* uses hash functions and the hashing process can take extra time. Memory requirements are determined by the size of data chunk that CaPC can process at a time. It depends solely on the implementation choice. The experimental results are generated by compressing and decompressing 4.3GB Wikipedia articles in XML format. For *LZMA*, *Gzip*, *Bzip* and *LZO*, compression levels are set to *-1* (fastest). The time and memory requirements include the *tar* process.

Also, note that compression ratio and performance varies depending on the contents of datasets. The above test results only give an idea on the characteristics of the CaPC. The test platform has the same configuration as the cluster nodes specified at the beginning of the section.

Table 3.3: Comparing CaPC with state-of-the-art compression algorithms. The original dataset is 4.3GB of Wikipedia articles in XML format. Compression levels for *LZMA*, *Gzip*, *Bzip* and *LZO* are set to *-1* (fastest).

| | Compression | | Decompression | | |
|---|---|---|---|---|---|
| Algorithm | Time(mins) | RAM(MB) | Time (mins) | RAM (MB) | Ratio |
| LZMA(5.1) | 1.87 | 13.36 | 2.11 | 2.91 | 96.6% |
| Gzip(1.4) | 1.04 | 1.81 | 0.68 | 1.79 | 93.6% |
| Bzip(1.0.6) | 14.52 | 7.95 | 1.54 | 5.09 | 91.1% |
| LZO(1.03) | 1.75 | 2.48 | 2.35 | 1.73 | 79.9% |
| CaPC(1.0) | 3.64 | ∼500 | 1.94 | ∼500 | 33.4% |

## 3.3 Conclusion

In this chapter, we presented the Content-aware Partial Compression scheme that, while not useful for small data, is effective for big data. Based on the observation that textual data often contains lengthy and repetitive strings, CaPC reduces the data size by replacing those strings with shorter codes. The overhead of the CaPC model files is trivial compared to the size of the data to be compressed. We evaluated CaPC using various standard MapReduce jobs and demonstrated its advantages in analysis performance, storage space, memory consumption and compatibility with existing software systems. Additionally, CaPC is particularly useful for repetitive analysis of large textual data, such as server logs, social media and web pages. One of the unique features of CaPC is that the partially compressed data can be used by existing algorithms and packages directly without modification.

CaPC is designed to make it effective for a variety of textual data analysis tasks. It assumes that the contents of the original data are irregular, for example, web pages and Twitter tweets, in which strings are separated by some application specific delimiters (e.g., white-space characters or colon characters), so that individual strings can be easily identified and replaced by shorter codes. Thus, it is not suitable for certain datasets, such as, genomic sequences, financial reports or sensor data, where the entire dataset may be considered as a single string or contains mostly numerical information. Also, because a string is simply replaced by a shorter code and the code does not contain any internal information about the string that has been replaced, sub-string searching is therefore impossible without referring to the compression model(s). However, frequently looking up the compression model(s) during analysis is highly discouraged as it will seriously impair performance.

# Chapter 4

# Approximated Huffman Compression (AHC)

*"Only those who will risk going too far can possibly find out how far one can go."*
— T. S. Eliot

Data processing and analysis often require flexible operations on data, for example, arbitrary string pattern searching. A word-based encoding scheme treats a word as the minimum unit. This limits the flexibility of querying arbitrary strings in a given text. A word-based compression also produces large size compression models in general. Referring to the architecture given in Chapter 3 (Figure 3.1), compression models need to be loaded from the HDFS Distributed Cache to each independent Map and Reduce nodes during the initialization phase of a MapReduce program. The larger model files make the initialization process slower. In contrast, a character-based encoding scheme exhibits flexibilities in string searching and produces very small compression models [1]. It generally implements a statistical method which is based on the principle that frequently occurring symbols can be represented using fewer bits. This implies a variable-length bit-oriented encoding scheme. There are several widely used bit-oriented encoders [Huf52] [Vit89] [LR81] [Eli75] [FK96] [Sal08] with each having different properties. Considering that our primary goal is to process compressed data in MapReduce, a successful candidate must fulfil the requirements that the encoded data can be directly modified and is splittable. The former requires a context-free scheme as discussed in Chapter 2 (Section 2.3.1). Among the existing encoding schemes, the Huffman algorithm [Huf52] is particularly suitable.

In general, splitting compressed data at arbitrary points will create the problem of

[1]In character-based compression, a model generally consists of all distinct characters (256 characters in ASCII encoding scheme) and associated frequencies. They can be organized in a very compact format and occupy very small space ($\sim$2KB).

broken codes which is also known as the code synchronization issue in information retrieval and telecommunication systems. Solutions to the problem include using synchronization points or using self-synchronizing codes. Ideally, self-synchronizing codes are preferred, since code boundaries are self identifiable or a broken code can be determined after examining a small number of consecutive codes. Previous research has attempted to obtain self-synchronizing codes from the Huffman algorithm to make the compressed data robust to the broken code issue without losing optimality. However, in general, Huffman algorithms can generate a set of valid codes for symbols based on a given probability distribution of symbols (compression model). But, the generated codes are not guaranteed to be self-synchronizing codes. According to Ferguson's theorem [FR84], self-synchronizing Huffman codes only exist when the probability distribution of symbols exhibit certain patterns. Given an arbitrary text, the probability distribution of symbols varies greatly. We need to adjust the probability distribution of symbols until they match the required patterns. This leads to a situation where an inaccurate probability distribution of the symbols is used with the Huffman algorithm to generate self-synchronizing codes which are very likely sub-optimal codes. This subsequently results in a poor compression ratio.

Using a synchronization point technique is another approach to deal with the broken code problem. We have discussed the drawbacks of using synchronization points in Chapter 2 (Section 2.3.2). The two main drawbacks are the space inefficiency due to frequently inserting extra information (synchronization points) and the difficulty of finding valid synchronization points that can be used across multiple files of a given dataset. To this end, we define a special packaging format that avoids the need for synchronization points and makes the Hadoop *Record Reader* component able to read a block of AHC compressed data efficiently.

Beside the coding issues, we also need to consider the efficiency of manipulating compressed data in MapReduce. Regardless of which encoding scheme is used, it is known that manipulating data at bit level is inefficient compared to that at byte level, especially for binary string searching and decoding. This is one of the main concerns when adopting bit representations for applications in data processing. To minimize the bit-operation inefficiency, we develop special data structures allowing a MapReduce program to efficiently and effectively process AHC compressed data without decompression.

Using bit-oriented encoding poses another concern regarding compression ratio. When the symbol richness [2] of a given text is relatively high, the average bit-code length tends to come close to the code length defined in the standard ASCII scheme. This will result in a low compression ratio. But, in fact, a substantial amount of the information on the Internet is produced by machines such as server logs and sensor outputs. These

[2]Symbol richness describes the number of distinct symbols (characters in the context of AHC) of a given text.

machine generated data files usually have fixed structure and low textual symbol richness. Other domain-specific data such as DNA/RNA sequences are very low in symbol richness. By employing a bit-oriented encoder, these types of data can be represented in a very compact format.

In this chapter, we present the AHC compression scheme that allows a MapReduce program to consume the AHC compressed data directly without decompression, while making the use of the compressed data transparent to developers. AHC is especially suitable for machine generated data and domain-specific data. In this work, we develop a hybrid data structure for AHC achieving $O(1)$ symbol decoding time. In addition, the AHC compression model can be used as a key to unlock the compressed contents and hence provides a certain level of protection on data privacy for data processing in public environments. The contributions of this work are:

1. the technique for upgrading the default Huffman tree to reduce sub-optimality introduced by the sampling processes;
2. the use of a lookup-table in conjunction with a conventional tree data structure to achieve constant decompression time;
3. providing supporting libraries for processing and manipulating data in compressed format in MapReduce.

## 4.1 Compression Scheme

AHC is based on the Huffman algorithm. It consists of two independent parts. The first part is the compression algorithm. It starts by building an approximated symbol probability distribution table through data sampling. Based on the sampling results (the probability distribution of symbols which is the compression model), we use the Huffman algorithm to generate bit-codes for symbols (ASCII characters). In addition, some enhancements are also made to the original Huffman algorithm in order to minimize the code sub-optimality under special circumstances as will be explained in Section 4.1.2. Specifically, we use a hybrid data structure for decoding a symbol in constant time. The second part of AHC includes the extension packages for Hadoop. The goal is to provide transparency between MapReduce programs, HDFS and the AHC compressed data. As a system, AHC follows the simple model illustrated in Chapter 3 (Figure 3.1). Both the compressed data and the compression model must be stored in HDFS and HDFS Distributed Cache, respectively. A MapReduce program uses the AHC supplied library functions to facilitate reading, processing and writing data in compressed format without decompression.

### 4.1.1 Compression Model

AHC is a statistical compression scheme based on the principle that a symbol with a higher probability of occurrence can be represented by fewer bits, subject to the principles of Shannon's theorem [Sha48]. Recall that a statistical compression method often comprises of three phases: *Transformation*, *Modeling* and *Encoding*. The transformation phase is selective and it involves reordering data. It does not compress data. However, the data reordering process makes the further compression process more effective when capturing and eliminating repetitive information, thereby a higher compression ratio can be achieved. For example, *Bzip* [Sew00] employs the Burrows-Wheeler Transformation [BW94] technique for data transformation at block level. As information is reordered in the block, direct access to *Bzip* compressed contents is extremely difficult. This can be overcome with the help of self-index techniques such as FM-Index [FM00], AF-Index [FMMN04] or Huffman-FM-Index [GNS$^+$06]. But, the main limitation with self-indexing is that it does not allow modification of the compressed data. Several other issues related to transformation and self-index techniques were discussed in Section 2.2.1 and Section 2.3.3. Considering all these, AHC does not use transformation techniques.

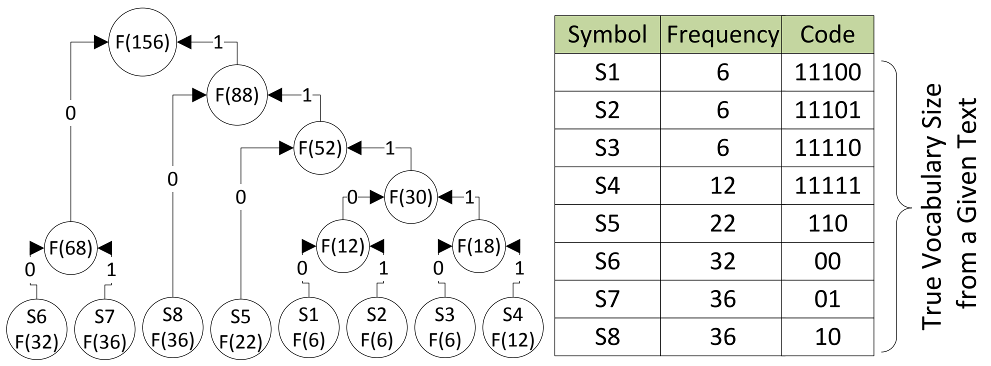
In Chapter 2, we have identified that using a context-free compression scheme allows one to manipulate compressed data freely without decompression. This requires AHC to work with either a *static* or *semi-static* compression model. Recall that a static method builds models based on prior knowledge or experience gained from compressing data having similar properties, for example, a pre-built codeword-map or a static dictionary. This can also be a subjective choice. It often results in poor compression ratios and is rarely used by modern compressors. Building a semi-static model requires scanning through the data contents in order to gain an insight into the symbol probability distribution of the data. The conventional semi-static compression algorithms use a two-pass approach to gather statistical information from an input dataset. The first pass performs a full-scan of the given dataset and uses the second pass to carry out actual data compression. In the big data context, a given dataset often has very large volume on the scale of gigabyte, terabyte or even petabyte. Scanning through such large scale data leads to an inefficient algorithm. It also poses a technical problem. When scanning big data, frequency of popular symbols can grow beyond the boundaries of a given data type defined in programming languages (e.g., an integer data type defined in Java has maximum value of $2^{31}$). In practice, many compressors perform symbol scanning on a per block basis, for example, *Gzip* and *Bzip*. This results in dedicated models per block. Recall that the main goal of AHC is to allow a MapReduce program to process compressed data without decompression. If models are built on a per block basis, the intermediate outputs from different Mappers will be encoded differently for the same information. This will cause confusion at the Reducers. For this reason, AHC must build a single compression model for the entire dataset(s). Rather than gathering precise information on symbol probability distributions through a full-scan, approximate information via sample statistics is more appropriate. AHC employs the sampling strategy explained in Chapter 3 (Section 3.1.3).

Once the symbol probability distribution table is built, the corresponding codes can be produced from Huffman algorithms. The compression process is simply replacing symbols by their corresponding Huffman codes. However, special care must be taken when using sampling. When performing a full-scan of a given dataset, we can collect exact information about which symbol has occurred. In contrast, a sampling process can not guarantee this. It is very likely that some rare symbols will be missed, especially when the sampling rate is relatively low. If the missing symbols are encountered during compression, it will lead to the termination of the entire compression process. It is often referred to as the zero-probability problem. Note that in a bit-oriented character-based compression, each character must have a valid code associated with it. A set of codes generated from Huffman algorithms are said to be prefix-free codes which guarantees no ambiguity between codes when decompression starts from the beginning of the compressed data in a sequential manner. This also implies that all information of a given text must be compressed. We can not treat informational and functional contents separately as done in the CaPC scheme introduced in Chapter 3.

### 4.1.2 Huffman Tree Upgrading

In order to fix the zero-probability problem, all possible symbols from both the standard (0 $\sim$ 127) and the extended (128 $\sim$ 255) ASCII codes must be present. A minimum frequency count (one) must be given to each symbol as an initial value as suggested in [WMB99]. After the sampling process, the observed symbols have frequency counts greater than one and the others are in place as a precaution. Eventually, all of the symbols will join the process of building a Huffman tree.

Statistically speaking, those non-observed symbols ($\Sigma^{S^-}$) have much lower probability of occurrence in the source data. In the situation where some of the observed symbols have relatively low frequency counts, more precisely, when a symbol's frequency counts $f_s$ are less than the cardinality of the $\Sigma^{S^-}$ , ($f_s < |\Sigma^{S^-}|$), then the non-observed symbols will affect the bit-code length of those observed symbols. This is illustrated by the following example. Given a text $T$ drawn from $\Sigma$, with a full-scan on $T$, symbols occurring in $T$ and their corresponding frequencies $F(s_i)$ can be accurately detected. Following the Huffman algorithm, starting with pairing the symbols having the lowest frequencies first, we can build up the Huffman tree for $T$ as shown in Figure 4.1 (*left*). In this example, we use the path convention of left "*0*" and right "*1*". Following the tree branches, we can retrieve the Huffman codes for each symbol occurring in $T$ as shown in Figure 4.1 (*right*). These are entropy codes. Based on this Huffman table, we can



| Symbol | Frequency | Code |
|---|---|---|
| S1 | 6 | 11100 |
| S2 | 6 | 11101 |
| S3 | 6 | 11110 |
| S4 | 12 | 11111 |
| S5 | 22 | 110 |
| S6 | 32 | 00 |
| S7 | 36 | 01 |
| S8 | 36 | 10 |

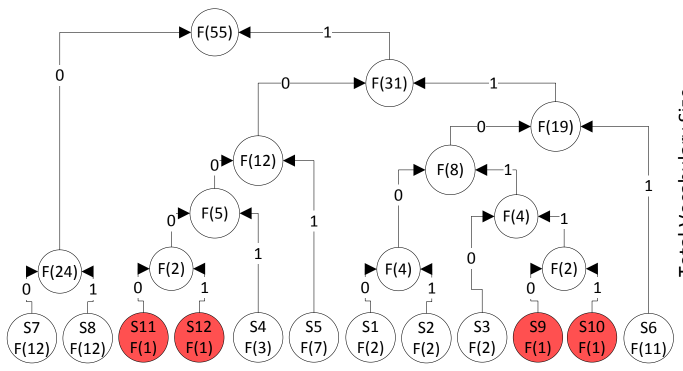
Figure 4.1: An example of a Huffman code table generated based on a semi-static compression model through a full-scan.

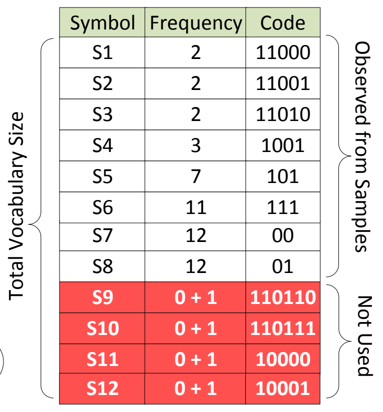
calculate the length of the coded text $H(T)$ using Equation 4.1, where $\ell$ denotes the length of the codewords.

$$H(T) = \sum(\ell * F(s_i)) \tag{4.1}$$

Referring to the statistics given in Figure 4.1, the Huffman compressed text for this example is 424 bits. The original $T$ is 1248 bits ($\ell = 8$ for ASCII codes). The compression ratio is therefore given by $\varphi \approx$ 66%. If the symbol frequencies are gathered from the sampling process, all possible symbols from alphabet $\Sigma$ must be present to avoid the zero-probability problem. In the above example, if we set $|\Sigma|$ = 12 (assume that characters in text $T$ drawn from 12 distinct characters to keep this example simple and clear), $|\Sigma^{S^+}| \bigcup |\Sigma^{S^-}| \equiv |\Sigma|$, where $\Sigma^{S^+}$ indicates the symbols that are observed from sampling of the text, according to [WMB99], for any $s_i \in \Sigma^{S^-}$ denoted by $s_i^-$, $F(s_i^-)$ = 1. The results make the $s_i^-$ participate directly in the Huffman tree construction process shown in Figure 4.2 (*left*). Assuming that the sampling results can truly reflect the population distribution of symbols, the codewords generated from the new Huffman tree are shown in Figure 4.2, note that the last four symbols shown in the table on the right is part of the alphabet $|\Sigma|$, but they are not used in the given text $T$. The $H(T)$ is now 444 bits.

To minimize this sub-optimality, we need to upgrade the frequency of the symbols that have been observed from the sampling process, to a certain level by multiplying the observed frequencies by a factor $\alpha$ defined in Equation 4.2, where $min(F(s_i))$ is the lowest frequency count for a symbol as observed from samples. The upgrading process is equivalent to building a Huffman tree for all observed symbols with a specially inserted pseudo-symbol as shown in Figure 4.3 (*left*), denoted by $S(H)$. All these *hapax*




Total Vocabulary Size
Observed from Samples
Not Used

| Symbol | Frequency | Code |
|---|---|---|
| S1 | 2 | 11000 |
| S2 | 2 | 11001 |
| S3 | 2 | 11010 |
| S4 | 3 | 1001 |
| S5 | 7 | 101 |
| S6 | 11 | 111 |
| S7 | 12 | 00 |
| S8 | 12 | 01 |
| **S9** | **0 + 1** | **110110** |
| **S10** | **0 + 1** | **110111** |
| **S11** | **0 + 1** | **10000** |
| **S12** | **0 + 1** | **10001** |

Figure 4.2: An example of Huffman code table generated based on a semi-static compression model through sampling.

*legonemas*[3] (non-observed symbols) are placed underneath the pseudo-symbol. This ensures that the inclusion of these *hapax legonemas* does not affect the length of the codewords generated for those symbols that have been observed from samples. Referring to the upgraded Huffman tree as shown in Figure 4.3 (*right*, note that the last four symbols showing in the table are part of the alphabet $|\Sigma|$, but they are not used in the given text $T$), we have $|\Sigma^{S^-}| = 4$, $min(F(s_i)) = 2$, thus $\alpha = 2$. Based on the upgraded codewords, the size of the compressed text is reduced to 438 bits now.

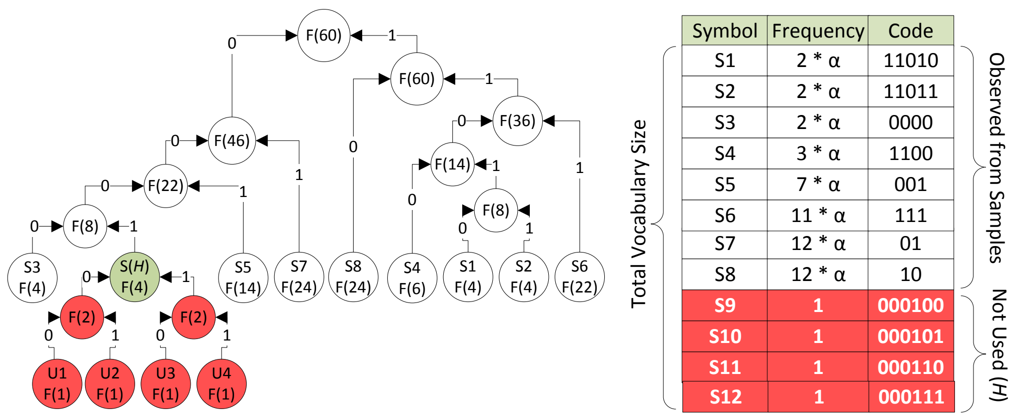
$$\alpha = \left\lceil \frac{|\Sigma^{S^-}|}{min(F(s_i))} \right\rceil \tag{4.2}$$

The upgrading process does not rely on how accurately the samples reflect the true population distributions, but it depends on the coverage of the captured symbols. If some rare symbols are missed in the sampling process, they will be placed underneath the pseudo-symbol. These missed symbols will have longer codewords than expected.

## 4.1.3 Algorithm Efficiency

The conventional Huffman algorithm is an entropy coder. Given a text $T[1,n]$ drawn from the alphabet $\Sigma = \{s_1, s_2, \cdots, s_m\}$ (distinct symbols occurring in $T$), according to Shannon's Theorem [Sha48], the average number of bits needed to encode a symbol in $\Sigma$ is given by Equation 4.3, denoted by $H_0$ (entropy), when a static or semi-static compression model is used with the Huffman algorithm to produce codes. The subscript

[3]Hapax legonema is a terminology commonly seen in the field of computational linguistics and natural language processing, meaning that a symbol only occurs once in a given text. It is a transliteration of Greek.



| Symbol | Frequency | Code |
|---|---|---|
| S1 | 2 * α | 11010 |
| S2 | 2 * α | 11011 |
| S3 | 2 * α | 0000 |
| S4 | 3 * α | 1100 |
| S5 | 7 * α | 001 |
| S6 | 11 * α | 111 |
| S7 | 12 * α | 01 |
| S8 | 12 * α | 10 |
| S9 | 1 | 000100 |
| S10 | 1 | 000101 |
| S11 | 1 | 000110 |
| S12 | 1 | 000111 |

Figure 4.3: An example of the upgraded Huffman code table generated based on a semi-static compression model through sampling.

$_0$ indicates a context-free compression. $n$ denotes the number of characters in $T$, $m$ indicates the number of distinct symbols in $\Sigma$. The space efficiency of the Huffman algorithm in compressing $T$ is given by $\Theta(n \cdot H_0(\Sigma))$.

$$H_0(\Sigma) = -\sum_{i=1}^{m} P_r(s_i) \cdot log_2 P_r(s_i) \tag{4.3}$$

AHC is based on the Huffman algorithm. It has similar space efficiency properties to the conventional Huffman algorithm, but, due to the upgrading effects, the space efficiency of AHC is conditional. The probability of symbol $P_r(s_i)$ is conditional on whether symbol $s_i$ is observed by the AHC sampling process or added as a child to the pseudo-symbol, as shown in Equation 4.4, where $F(s_j^+)$ indicates the frequency counts of the symbol $j$ observed in samples drawn from $T$, $|\Sigma^{S^-}|$ is the number of non-observed symbols in $\Sigma$.

$$P_r(s_i) = \left\{ \begin{array}{ll} \dfrac{\alpha \cdot F(s_i)}{\alpha \cdot \sum_{j=1}^{|\Sigma^{S+}|} F(s_j^+) + |\Sigma^{S-}|} & \text{if} \;\; s_i \in \Sigma^{S+} \\ \dfrac{1}{\alpha \cdot \sum_{j=1}^{|\Sigma^{S+}|} F(s_j^+) + |\Sigma^{S-}|} & \text{if} \;\; s_i \in \Sigma^{S-} \end{array} \right\} \tag{4.4}$$

Additionally, AHC uses a block-based packaging scheme to avoid the broken code problem. The packaging format for blocks requires extra bits. This will be elaborated in detail in Section 4.1.4. Referring to Figure 4.7, each AHC-Block occupies one full byte (8 bits) used as an indicator byte. Each AHC-Block may also insert a variable number of extra bits at the end of the block (before the indicator byte) to make the entire AHC-Block contain a whole number of bytes. In the worst case scenario, the number of trailing bits needed is equal to the longest codeword minus one ($max(\ell) - 1$) where $\ell$ is the length of a codeword. Thus, the AHC space efficiency for compressing text $T$ containing $n$ characters, is given by Equation 4.5, where $b^{AHC}$ indicates the size of the AHC-Block.

$$\begin{aligned} O(n \cdot H_0^{AHC}(\Sigma)) &\Rightarrow O(n \cdot H_0(\Sigma) + \left\lceil \frac{n}{b^{AHC}} \right\rceil \cdot 8 + \left\lceil \frac{n}{b^{AHC}} \right\rceil \cdot (max(\ell) - 1)) \\ &\Rightarrow O(n \cdot H_0(\Sigma) + \left\lceil \frac{n}{b^{AHC}} \right\rceil (max(\ell) + 7)) \end{aligned} \tag{4.5}$$

The time efficiency of the Huffman algorithm using a conventional method [4] is given by $\Theta(|\Sigma| \cdot log_2(|\Sigma|))$. This is the time complexity for constructing a Huffman tree for $|\Sigma|$ distinct symbols. It should not be confused with the time complexity of encoding and decoding Huffman compressed data. Once the Huffman tree is built, the data encoding time using the Huffman algorithm is given by $O(n)$.

The decoding performance has a big influence on pattern searching, counting and extraction. This is one of our major concerns when manipulating the AHC compressed data in MapReduce. Generally, the decoding time is determined by the depth of the tree. Further improvement has been made by [Has04] taking $\Theta(n \cdot t)$ time, where $t$ is the number of distinct codeword-lengths and $n$ is the number of characters in a given text. The most recent research improves the Huffman decoding performance to $\Theta(n \cdot \lceil \frac{max(\ell)}{z} \rceil)$ [LHY12] by extracting multiple symbols at a time. The improvement is conditional on $min(\ell) > \frac{z}{2}$, where $z$ is defined as the size of the processing unit. The main limitation of this method is the complexity of constructing the decoding data structures by splitting a Huffman tree into a number of sub-trees and thereafter merging the sub-trees to form a valid recursive Huffman tree. Furthermore, specialized binary string searching algorithms are also studied in the literature [SD06] [FL08], in which the q-Hash algorithm [FL08] achieves a time complexity of $O(n \cdot \left\lceil \frac{m}{q} \right\rceil)$, where $m$ is the length of the binary string pattern to be searched and $q$ is the length of the sub-string of the pattern. However, the cost of the preprocessing time and the high complexity involved in constructing the decoding data structures associated with the aforementioned algorithms makes them less practical in general.

AHC employs the lookup-table approach used by the open-source *Gzip* compressor [lGA] [ZL77] [Deu96] with a fine adjustment to it. A lookup-table is essentially a static codeword to symbol map in which a codeword is the index in the lookup-table. So that, given a codeword, its corresponding symbol can be found in $O(1)$ time. In

[4]The conventional method constructs a Huffman tree using a heap data structure. Each distinct symbol is a leaf in the heap and the property of the heap data structure needs to be maintained after each symbol insertion by keeping the symbol having the largest probability value at the higher level. The heapifying processes dominate the time consumption for the Huffman algorithm which is affected by the number of leaves in the tree [SW14].

*Gzip*, data is compressed on a per block basis. Each block is compressed using a combination of *LZ*77 [ZL77] and the Huffman algorithm. The important part is that the Huffman trees [5] are built specifically for each block (there are two Huffman trees per block [Deu96]). During decompression, the lookup-tables will be constructed based on the individual Huffman tree for each block. When the size of the compressed data is large, the accumulated time used for the lookup-table construction is considerably long. This forces *Gzip* to seek a balance between the performance and the lookup-table size. *Gzip* uses hierarchical lookup-tables. By default, *Gzip* uses $2^9$ (512 entries) for the first level lookup-table. In contrast, AHC is required to build a single compression model for the entire dataset(s), so that multiple files can share the same compression model in MapReduce in a distributed environment. Recall that each Mapper receives a block of data, thus the lookup-tables only need to be constructed once for each Mapper. And the size of the data block (128MB in default configurations of HDFS) is much larger compared to the block size used in *Gzip* (depending on the internal buffer usage and the efficiency of the current Huffman trees [Deu96]), therefore, AHC does not lose performance due to frequently constructing lookup-tables as seen in *Gzip*. In AHC, we can use much larger lookup-tables. As a result, AHC can be much faster than using the traditional Huffman trees in decompression.

AHC uses $2^{12}$ (4096) entries for the lookup-table. Codewords longer than 12 bits will be decoded using a Huffman tree. That is, the lookup-table covers all possible combinations of 12 bits. Any codeword having length less than or equal to 12 bits will be decoded using the lookup-table. This is a much bigger table than the one used by *Gzip*. It consumes more system memory. But, considering modern clusters are often configured with large amounts of memory, the memory required for the decoding data structure is trivial. The main reason that drives us to use both the lookup-table and Huffman tree is the complexity and time for constructing hierarchical lookup-tables. In *Gzip*, if a codeword is longer than nine bits, a hierarchical lookup-table(s) will be built. How many levels of lookup-table are needed depends on the variations of the codewords. Recall that *Gzip* builds Huffman trees on a per block basis and the variation of the symbols is confined to each block. Additionally, *Gzip* terminates a block when the current Huffman trees become inefficient. This makes the process of constructing lookup-table(s) much lighter. In contrast, AHC uses a single Huffman tree for the entire dataset(s). All possible symbols must be included when building a Huffman tree. For efficiency and complexity reasons, we use a single lookup-table to cover the majority of the symbols and use a Huffman tree for decoding long codewords. Probabilistically, a symbol having a longer codeword has less chance of occurring in a given text. Thus, AHC still can achieve constant time decoding. We use an example to explain how the lookup-table and Huffman tree are used together for AHC decompression. The details

[5]Huffman tree is a data structure built based on a given compression model (probability distribution of symbols). It will be used for compressing and decompressing data.

are illustrated as follows.

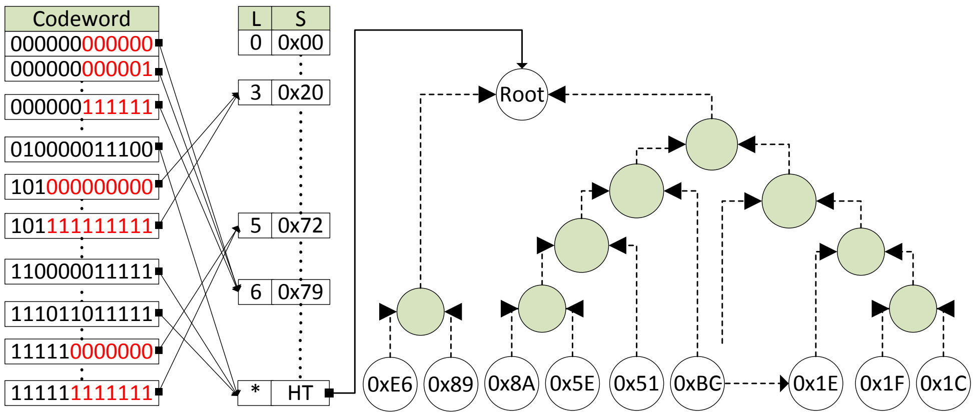


Figure 4.4: An illustration of AHC lookup-table with Huffman tree.

In Appendix B, we have codewords generated for 2GB of *Wikipedia articles* (dataset: *DS-WikiEN*). There are in total 256 symbols in the standard and extended ASCII table. 237 of them are included when building the hybrid data structure. 19 of them are eliminated as listed in Appendix B (Table B.2) because these symbols are never used in text files. Removing the unused symbols can reduce the maximum length of codewords. If we adopt the *Gzip* approach by building multiple lookup-tables, then, for the list of codewords as shown in Appendix B (Table B.1), there will be at least 12 lookup-tables organized hierarchically (lookup-tables can have different sizes). In AHC, we build a single lookup-table of size $2^{12}$. The lookup-table is filled up by all combinations of the $2^{12}$ binary sequences. In order to identify the codeword length and the symbol associated with the codeword, we need to do codeword masking. For example, referring to Appendix B (Table B.1), codeword "*000000*" is assigned to symbol "*y*" (*0x79*) and has length 6 bits. We take a 12-bit sequence from the input data and logical AND the 12-bit sequence with a 12-bit mask of "*111111000000*", and if the first six bits of the 12-bit sequence are "*000000*", then regardless of the other six bits, we are certain that the first symbol of the 12-bit sequence is the symbol "*y*" with codeword length six. That is, the codewords in the range [000000000000, 000000111111] point to the same symbol "*y*". For the same reason, when we read 12 bits from any input data and convert them into a numerical value which will become the index of the lookup-table, then the first symbol can be found in one operation. The lookup-table index size (12 bits) can be seen as the minimum processing unit. Subsequently, the input data can be advanced six bits and the decoding of the next 12 bits can begin.

In another example, to decode a given bit sequence of "*10100000010111111110001010*" as shown in Figure 4.5, we need to read the first 12 bits from the given bit sequence and convert them into a numerical value (2565) which is the index into the lookup-table.

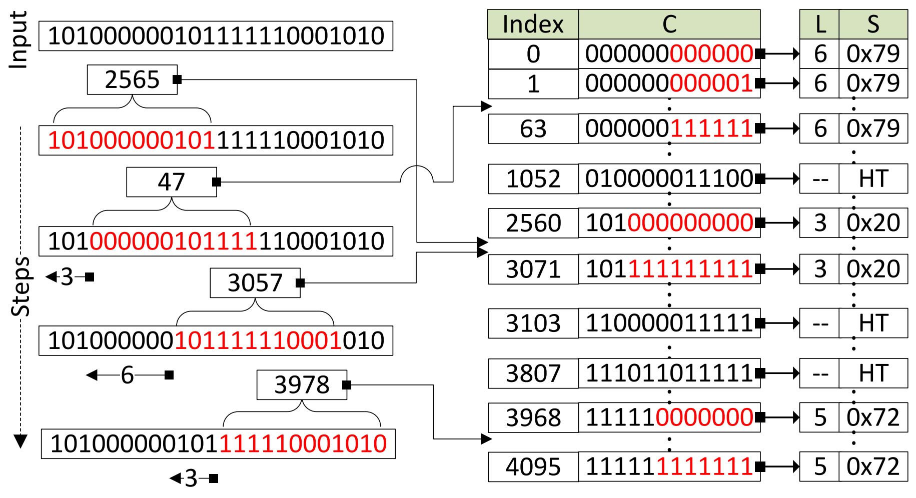


Figure 4.5: An example of AHC lookup-table.

The index is between $[2560, 3071]$ which is the range generated from the codeword "*101XXXXXXXXX*" masked with "*111000000000*". The index points to the corresponding symbol and the codeword length. Based on the codeword length, we can advance the input three bits and start decoding the next consecutive 12 bits. The decoding steps are shown in Figure 4.5. This does not cause any codeword confusion because the Huffman algorithm generates prefix codes. This means that a shorter code is never going to be a prefix of any longer code in the same Huffman tree.

If a codeword is longer than 12 bits, for example, the symbol "@" (*0x40*) has codeword "*01000001110011111*" and length 17. The first 12 bits can be found in the lookup-table, however the associated fields do not contain a valid symbol, but point to a Huffman tree for decoding. This makes AHC decoding time conditional. Decompressing text $T[1, n]$ is therefore conditional on the length of the codewords expressed in Equation 4.6, where $s \in \Sigma$, $t$ denotes the lookup-table size. Because we use the lookup-table to cover $1 \sim 12$ bit patterns inclusive, the most frequently occurring symbols will be found directly from the lookup-table in one operation, thus the AHC decompression time is very close to $\Theta(n)$.

$$n \cdot (\sum_i (P_r(s_i)) + \sum_j P_r(s_j) \cdot \ell(s_j)) \quad \{i, j | \ell(s_i) \leqslant t, \ell(s_j) > t\} \tag{4.6}$$

It is assumed that the samples reflect the true population distribution of symbols. Referring to statistics given in Appendix B (Table B.1), there are 91 symbols having codeword length less than or equal to 12. The probability of symbols can be calculated by their corresponding frequency counts. We have $i = 91$ and

$\sum_{i=1}^{91} P_r(s_i) = 0.9959750685739751$; $j = 237 - 91 = 146$, and $\sum_{j=1}^{146} P_r(s_j) \cdot \ell(s_j) = 0.05645490855757191$. Inserting these numbers into Equation 4.6, we get $1.05n$. Thus, for this particular example, the time complexity grows asymptotically as fast as $n$ and it is given by $\Theta(n)$. Using the conventional Huffman tree to decode the compressed text will take $n \cdot \sum_{j=1}^{237} P_r(s_j) \cdot \ell(s_j)$ time. For the same example, this gives $5.19n$, which is 79.8% slower than the AHC.

**Algorithm 1:** AHC pattern searching

```
Data: $P \longleftarrow$ Pattern to be searched in plain string format.
Data: $S \longleftarrow$ Source data in AHC compressed format.
Result: Index of the first found or "-1" if not found.
begin
    /* Transform P from string to AHC compressed binary */
    $P_b \longleftarrow AHC.T(P)$
    if $(P_b.length > S.length)$ then
        return -1
    $idx \longleftarrow 0$
    while $(idx < S.length - P_b.length)$ do
        if $(P_b \oplus S[idx,\ idx + P_b.length])$ then
            $idx \mathrel{+}= lookup(idx)$
        else
            return $idx$
    return -1
```

The lookup-table is mainly used for decoding symbols or finding how many bits need to be advanced during pattern searching. In AHC, all input strings will be transformed to binary bit sequences. For example, if we want to search for string "*data*" in an AHC compressed bit sequence of "*1100100*0011*011101*101*00001*0101*1000*0101*" (referring to Appendix B <Table B.1>, the given binary sequence corresponds to the string "*big data*"), we need to transform the pattern "*data*" into the AHC coded bit sequence "*00001*0101*1000*0101*". Note that the asterisk characters are used to split the codewords for clear presentation. They are not used in the actual compression. The search procedures for the binary sequences are given by Algorithms 1 and 2.

From an algorithmic point of view and ignoring the inefficiency of operations at the bit level of the underlining system, the pattern searching efficiency is also given by $\Theta(m(n \cdot (\sum_i (P_r(s_i)) + \sum_j P_r(s_j) \cdot \ell(s_j)) - m))$. When the majority of symbols have length less than the lookup-table size, the AHC searching complexity is also $\Theta(m(n-m))$.

### 4.1.4 AHC Block Packaging

Hadoop is a distributed environment. Data stored in HDFS is divided into blocks and distributed across cluster nodes. A MapReduce program takes advantage of this dis-

**Algorithm 2:** AHC table lookup

```
$LT \longleftarrow$ initialize lookup-table.
$HT \longleftarrow$ initialize Huffman tree.
$t \longleftarrow 12$                                /* Processing unit bit length.  */
Function lookup($pos$)
    Data: $S$ : Source data.
    Data: $\ell$ : Codeword length.
    Data: $pos$ : Current position of S.
    Result: Codeword length.
    if ($LT[S[pos, pos+t]] \mapsto codeLength$) then
        $return\ \ T[S[pos, pos+t]] \longrightarrow \ell$
    else if ($T[S[pos, pos+t]] \mapsto HT$) then
        $currentNode \longleftarrow HT.root$
        while ($currentNode \neq HT.leaf$) do
            if ($S[pos+\ell] = 0$) then
                $currentNode \longleftarrow HT.left$
            else if ($S[pos+\ell] = 1$) then
                $currentNode \longleftarrow HT.right$
            $\ell{+}{+}$
    $return\ \ \ell$
```

tributed data storage by processing data blocks in parallel. This requires that the AHC compressed data is splittable to the HDFS.

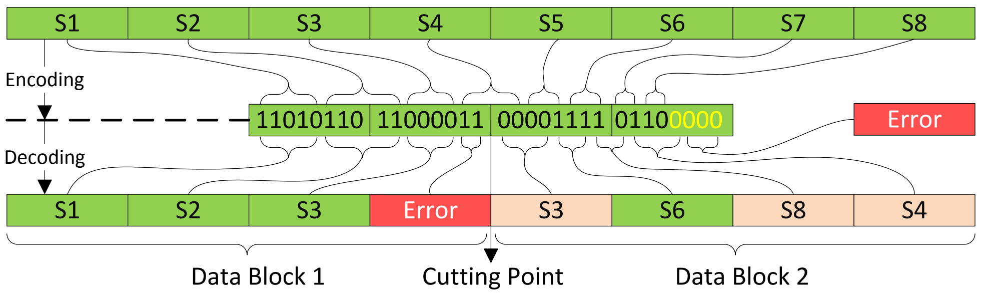


Figure 4.6: An illustration of broken codes in AHC.

Recall that HDFS splits data into a series of fixed-size data blocks (HDFS-Block). The data block size is chosen for the best disk I/O performance as explained in Chapter 2. Each data block contains a whole number of bytes. Note that AHC is a variable-length encoding scheme. Breaking AHC compressed data at an arbitrary point will highly likely create a broken code at the data boundary. An example is shown in Figure 4.6, using the codewords showing in Figure 4.3. In this example, the AHC compressed data is split in the middle of the codeword generated for symbol "$S4$". After splitting the compressed data, each data block will be assigned to and processed by a dedicated Mapper. Mappers are independent of each other. This means the Mapper that processes

DataBlock-2 does not know whether the symbol "*S3*" was correctly decoded or not. In general, the position of the cutting point is determined by the configured HDFS-Block size. In our experimental environment, the HDFS was configured with a 128MB block size. Thus, at least, we need to prevent the broken code problem from happening at every 128MB data boundary in the AHC compressed data.

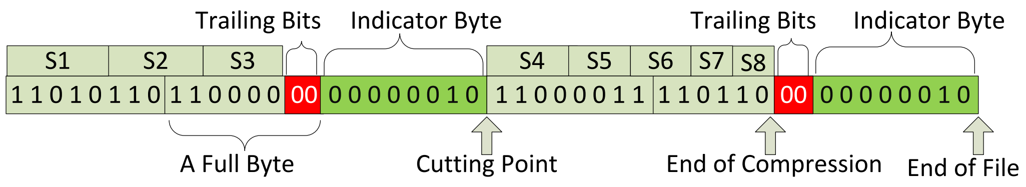


Figure 4.7: AHC block-packaging scheme.

In fact, letting each Mapper process 128MB data as a whole at a time is inefficient. More often in the MapReduce framework, each HDFS-Block is further broken down into smaller records which are often specific to the dataset and/or application domain. The *Record Reader* component of the MapReduce framework is responsible for parsing and fetching records to Mappers. In general, each Mapper will process a single record at a time. Considering that records can have variable length, taking care of the broken code problem for every single record is inefficient in terms of compression ratio and speed. To this end, we develop a fixed-size block packaging scheme as illustrated in Figure 4.7. That is, the *Record Reader* of MapReduce reads a fixed block of data and fetches the data block to the Mappers. Mappers are responsible for parsing records from the data block received.

By default, AHC uses a fixed-size block of 512KB (AHC-Block). The last byte of the AHC-Block is reserved and used as the Indicator Byte which indicates how many trailing bits have been inserted to make the AHC-Block a whole number of bytes. Additionally, the HDFS-Block size must be divisible by the AHC-Block size. The Indicator Byte occupies a full byte. It can indicate up to 255 Trailing Bits. This is sufficient to cover the worst case scenario, when the model is left or right skewed, for example, when the probability of the 256 possible symbols grows in a Fibonacci sequence [Sal08]. Besides the splittable requirement, the compression model file must be available to all data blocks. A conventional compressor would put a compression model as the header of the compressed file. In contrast, we need to store the compression model and the compressed contents in separate files.

### 4.1.5 AHC Hadoop Extension

Recall that, using the AHC scheme, we have shifted the responsibility of parsing records from the *Record Read* component to each individual Mapper. To be able to break a data block (AHC-Block) into *records* at the *Map* stage and write compressed output at the *Reduce* stage, customized *Input Format*, *Record Reader* and *Output Format classes* [6] are needed as extensions to Hadoop. In addition, AHC is a bit-oriented and variable-length encoding. Each record must be held by a bit-set [7] like data structure as implemented in the *BBP_BitSetWritable class* for the AHC scheme. This *class* must implement all equivalent operations found in the standard Java *String* class. Other utility classes are also implemented to make operation transparency between developers and the compressed data.

For many data analysis jobs, parsing data records is necessary. With clear text, manipulating data is simple as the data content is readable by developers. In AHC, we provide a transformation function, *AHC_T()*, that converts between original text and AHC compressed data. For example, if we write a regular Java clause for searching a phrase "*value.contains("big data");*", in AHC, it is simply "*value.contains(AHC_T("big data"));*". In this example, the variable "*value*" in the regular program can be a String data type. In AHC, the variable "*value*" needs to be a *BBP_BitSetWritable* data type. The "*contains()*" function in AHC is an implementation of the "*contains()*" function found in the regular Java String class for the *BBP_BitSetWritable* data type. The transformation from regular data-type to AHC bit-set or vice versa requires the AHC compression model file. This model file needs to be loaded at the Map or Reduce initialization phase so that the AHC hybrid data structure can be built and ready for data-type transformation. This may incur a small delay due to loading the model file from HDFS Distributed Cache to each computational node over the network. Note that the model file contains 237 characters and their associated frequency counts. The model file size is approximately 2KB.

## 4.2 Evaluation

We evaluate the effectiveness of AHC using an on-site Hadoop cluster. The characteristics of AHC and the impact of using AHC in MapReduce are demonstrated using standard MapReduce jobs summarized in Table 4.1. A list of real-world datasets used in the evaluation is shown in Appendix A (Table A.2). The cluster hardware specification and the topology of the experimental environment are given in Appendix A (Table A.1 and

[6]In object-oriented programming, "*a class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform*" [GHJV95].

[7]A bit-set data structure is also known as bitmap, bit array, or bit vector. It is essentially an array data structure that compactly stores bits. The size of the array can grow as needed.

Figure A.1). The cluster was configured with Linux kernel version 2.6.32 and Hadoop version 2.5.0. HDFS replication was set to one and the HDFS data block size was set to 128MB. The AHC default sampling rate is used across all experiments. Exceptions are explicitly indicated.

### 4.2.1 Performance

Table 4.1: A summary of MapReduce jobs used for evaluating the AHC compression scheme.

| | Jobs | | | |
|---|---|---|---|---|
| Parameters | Music Rank | | Finding Implant Sequence | |
| Dataset | *DS-Yahoo* | | *DS-mRNA* | |
| Data Type | Original | AHC | Original | AHC |
| Input | 10.9 GB | 4.9 GB | 13.2 GB | 3.7 GB |
| Intermediate | 4.5 GB | 4.4 GB | 11.6 MB | 4.7 MB |
| Output | 3.2 MB | 3.2 MB | 17.0 Bytes | 17.0 Bytes |
| Duration | 11m48s | 11m43s | 2m05s | 1m34s |
| Performance Gains | – | 0.1% | – | 24.8% |
| Size Reduction | – | 55.0% | – | 72.0% |
| *1GB = 1,073,741,824 Bytes* | | | | |

In the *Music Ranking* job, we use the *Yahoo! Music User Rating* dataset [Yah06] to rank $\sim$136 thousand songs from $\sim$700 million ratings given by $\sim$1.8 million users. The dataset only contains numerical values. Records are separated by *new-line* characters. Fields of each record are separated by *Tab* characters. The symbol richness of this dataset is relatively low. AHC compression reduced the data size by 55%. In this job, we observed no performance gains. This is due to two reasons.

1. Firstly, the dataset has low symbol richness. AHC achieves the same $O(m(n-m))$ time complexity on pattern searching as if it is operated at byte level. The inefficiency is mainly due to the data manipulation, such as binary string concatenation, at bit level. In more detail, when concatenating two strings at byte level, a new internal buffer long enough for two strings is created, then followed by copying the two string into the new buffer in order. At bit level, if the first binary string does not occupy a full number of bytes, the last several bits in the last byte of the first binary string need to be merged with the first several bits in the first byte of the second binary string. The merged result produces a full byte and all the successive bytes of the second binary string need to be left shifted accordingly. The shifting process creates a non-negligible overhead when the second binary string

is relatively long. This can be improved by employing indexing techniques which can be used to remember the concatenation point of the two binary strings, thus bit shifting operations can be avoided for the second binary string. The problem is that working with indexing techniques requires maintaining both the index files and the original files. This makes the total data size much larger.

2. Secondly, when Mappers receive data from the *Record Reader* component, the received data is organized as a *String* data-type. When comparing numerical values, if the original data is used, the data in the *String* data-type will be converted to corresponding numerical value data-type (e.g., *Integer*, *Float* and *Double*). If AHC compressed data is used, the compressed data needs to be decompressed first and then converted to the corresponding numerical value data-type (we can not directly convert from AHC compressed data to other data-types). This decompression process takes extra time. In addition, in order to perform the data type conversion, we also need to load the AHC compression model from the HDFS Distributed Cache to each Mapper and/or Reducers over the network, which is another extra step.

However, as AHC has reduced data size by 55%, loading the 55% smaller data from persistent storage (hard disks) to memory requires much less time. Also, considering that AHC achieves similar string searching complexity at bit level in compressed format compared to that at byte level in clear text format, thus searching strings (compressed binary strings) in AHC compressed data is faster than searching the same strings in the original text. Although, we do not get overall performance improvements, we have saved 55% storage space without sacrificing analysis speed.

In the *Finding Implant Sequence* job, we demonstrate a special case where AHC can save a surprisingly large amount of storage space. We chose to use a human *mRNA* (messenger Ribonucleic Acid) dataset [Con09]. Using a word-based compression approach, this would not be possible, since the entire dataset may be presented as a single string. The task is to find an implanted *mRNA* sequence in the given dataset(s) and count its appearances. It may be argued that *mRNA* can be represented in a 2-bit format since it only contains four valid characters {*C*, *G*, *A*, *U*}. This only applies to a dataset that has been preprocessed and cleaned. Generally, a *mRNA* sequence may contain upper-, lower-case and other characters, for example, character "*N*" for missing and *new-line* for segmentation, then the 2-bit representation can not be used. For this particular dataset, AHC reduces the data size by 72% and increases the performance of the MapReduce program by 24.8%.

In general, using the AHC compressed data requires much less system memory. This is because Mappers are independent processes, each of them has a dedicated *Java Heap Space* [8] assigned to it. The more Mappers the more aggregated memory is required.

[8]Hadoop MapReduce is implemented in the Java programming language. A MapReduce application

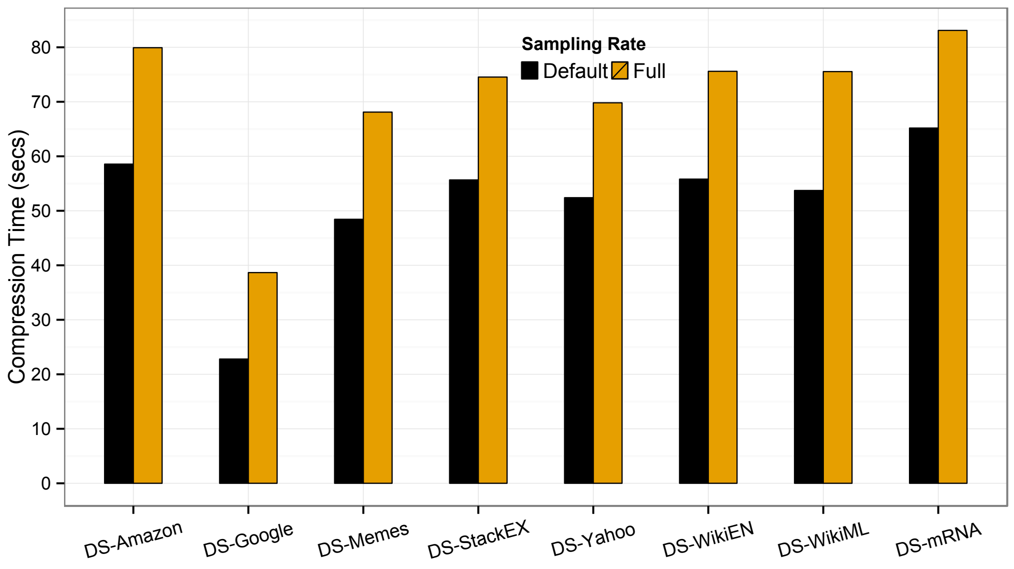


Figure 4.8: Comparison of AHC compression speed using sampling methods with the default sampling rate and full-scan methods. 2GB data from each dataset was used in the experiments.

Recall that the input data size is the dominant factor affecting the number of Mappers. After compression, the data size is significantly smaller than the original data. This leads to a smaller number of Mappers and consequently less memory consumption. Also, data structures used to store compressed binary strings in memory require much smaller internal buffers.

### 4.2.2 Effectiveness of Sampling

Scanning through large dataset(s) for statistical sampling is time consuming. To improve compression speed, the default sampling process uses a small fraction of the source (the default sample size for 2GB data is 1.8MB). The default sample size is logarithmically proportional to the file size. Figure 4.8 shows that using the default sampling rate can save up to 20 seconds (25%) compared to a full-scan on 2GB datasets. The difference is solely dependent on the original data size. When the original data size increases, the difference will become larger. In addition, we gradually increase sampling rate to examine its effects on compression ratio, as shown in Figure 4.9. For each dataset, with a carefully designed sampling strategy, a 5% sampling rate can accurately reflect the true population distribution for most datasets. From the default (1.8MB) to {5% (102.4MB) ~100% (2GB)} sample size, there is only an improvement of several hundred kilobytes. The fluctuation in the *mRNA* sequence is due to the random nature

is also a Java application that complies with the MapReduce programming paradigm. Java applications are only allowed to use a certain amount of memory (Java Heap Space) specified in the Java runtime environment.

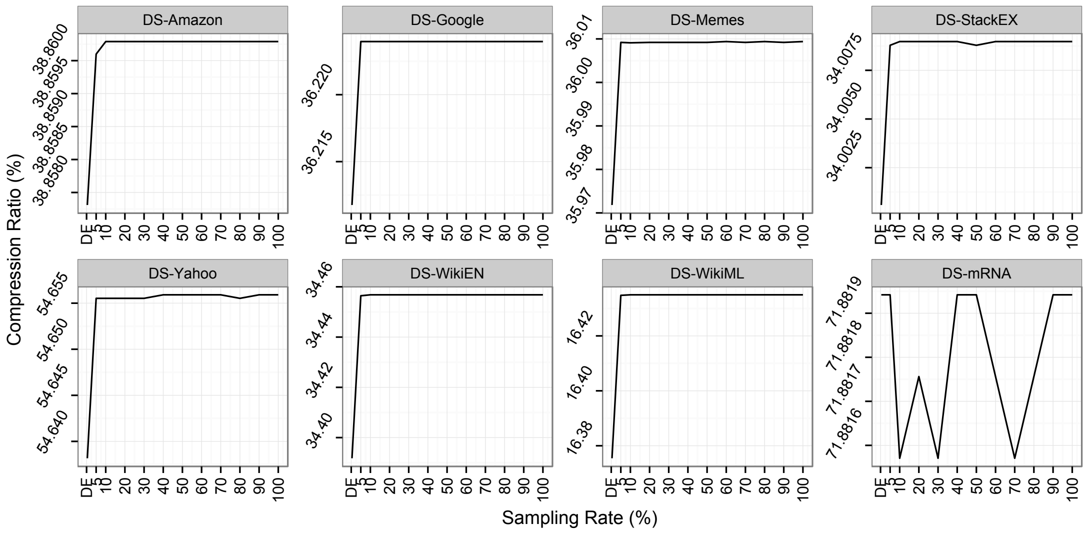


Figure 4.9: Sampling rate effects on compression ratio.

of the genome sequence.

### 4.2.3 Upgrading Effect

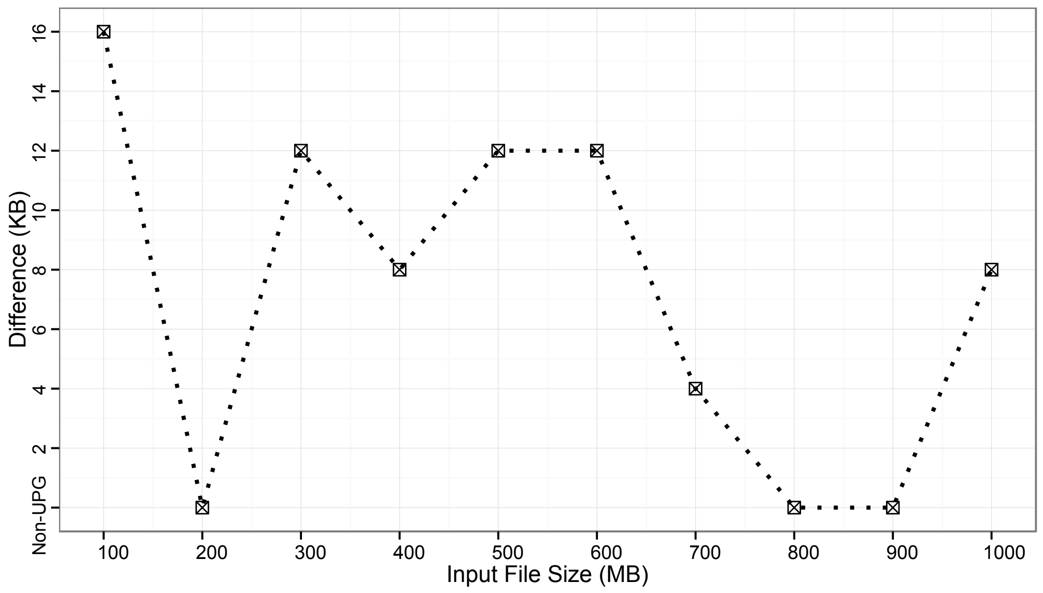


Figure 4.10: Huffman upgrading effects on compression ratio.

The effectiveness of the Huffman upgrading relies on the symbol coverage of the sampling. In this experiment, we draw a set of data from dataset *DS-WikiEN*. We compress each dataset with the upgraded Huffman and non-upgraded Huffman to identify the difference. We use a 5% sampling rate for all the tests in this section. Note that in the previous section, we learned that a 5% sampling rate can accurately reflect the true population distribution of symbols. In Figure 4.10, we can see that in the worst case scenario, using the upgraded Huffman tree can produce the same compression ratio as the non-upgraded Huffman tree. In other cases, the upgraded Huffman can further reduce data by several kilobytes. The bottom line is that if the sampling process can obtain all the symbols that have actually occurred in a given text, regardless of whether the symbol probability distribution is accurate or not, in the worst case scenario, the upgraded Huffman will produce the same compression ratio as if the conventional Huffman was used. In Figure 4.10, the difference shows no pattern as the data size increases. This is solely because of the randomness of the sampling and the source content.

### 4.2.4 Further Compression

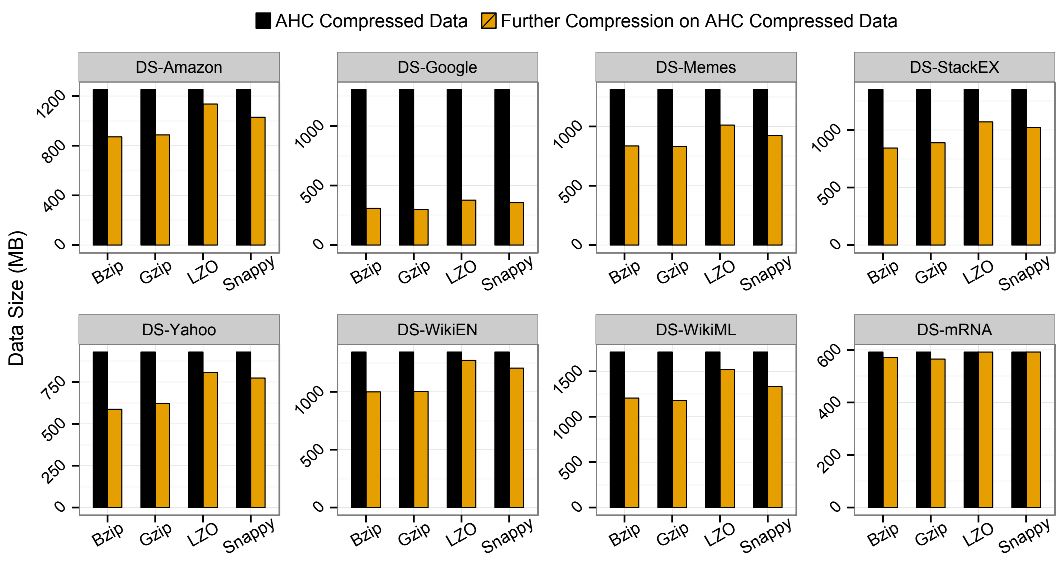


Figure 4.11: Evaluation of the impacts on applying a further compression to the AHC compressed data.

Hadoop allows compression of the input, intermediate and output data with options of using various compression algorithms, including *Gzip*, *Bzip*, *LZO*, *Snappy* and most recently *LZ4*. When using AHC with MapReduce, the data has already been compressed and the original pattern in the source data is lost. We evaluate whether there is value in applying further compression to the AHC compressed data, especially for the intermediate output data.

In this experiment, we use 2GB of data taken from each dataset. We compress the 2GB data using AHC, then apply other compression algorithms to the AHC compressed data. The results are shown in Figure 4.11. In general, we can still obtain several hundred megabytes further reduction in size. There are two extreme cases. For the *Google*

*Server Log* files (dataset: *DS-Google*), there is a significant size reduction after applying further compression. This is due to the repetitiveness of the original data. The AHC compression by definition only replaces symbols by shorter bit-oriented codewords. It does not break the repetitive nature of the original contents. By applying further compression, this repetitiveness can be captured by those general purpose algorithms which utilize context information for compression, hence this results in further size reduction. In contrast, applying further compression on the *mRNA* sequence does not have much effect, it is even slightly worse for *LZO* and *Snappy*. This is mainly due to the randomness of the contents. In principle, if a given dataset is known to contain very repetitive contents, it may be worth applying further compression.

### 4.2.5 AHC Characteristics

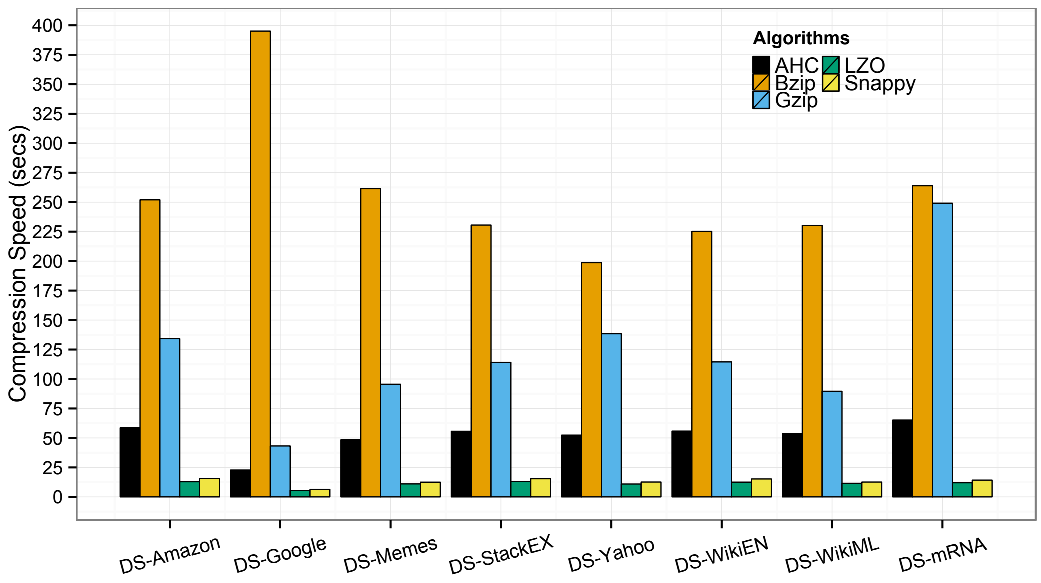


Figure 4.12: AHC compression speed comparing to Hadoop supported algorithms with the default compression level configured for *Bzip*, *Gzip* and *LZO*.

AHC has different goals to other compressors. To give a sense of compression/decompression speed and ratios, we compare AHC with four other compressors supported by Hadoop: *Gzip* (*v1.4*), *Bzip* (*v1.06*), *LZO* (*v1.03*) and *Snappy* (*v1.1.1*). For *Gzip*, *Bzip* and *LZO*, the compression level is set to *-1* (fastest). Time requirements include the *tar* [9] packaging process. Results are obtained from compressing 2GB of data taken from each dataset. In Figure 4.12, AHC shows relatively stable compression speed across datasets. After the sampling process, a Huffman tree is built and codes are generated for symbols. The compression process is then just finding a symbol-code mapping in a

[9] *tar* is a Linux tool that is used for collecting, distributing, and archiving files, while preserving attributes associated with those files. These attributes, for example, include user access permissions and directory structures.

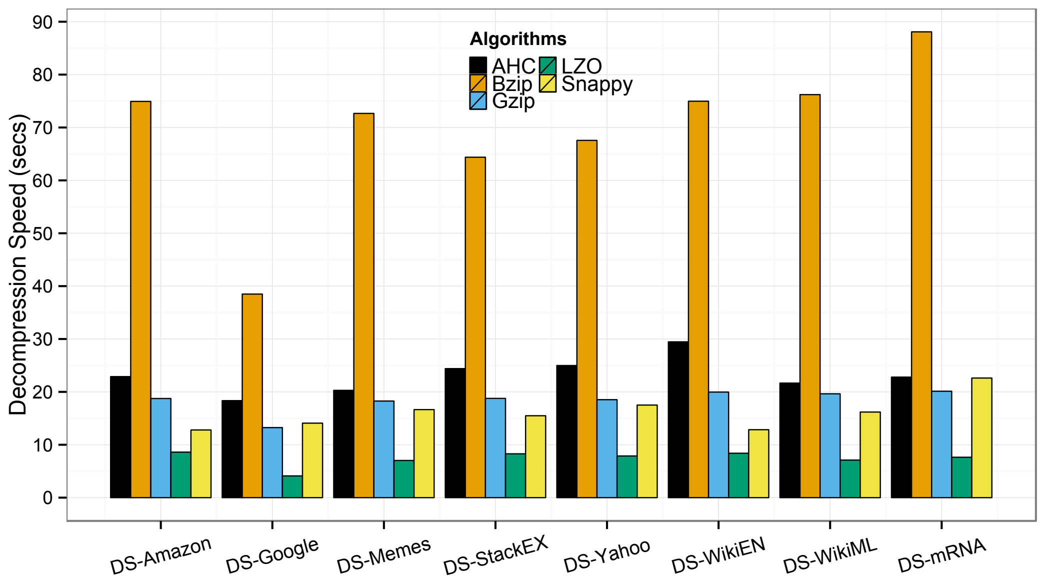


Figure 4.13: AHC decompression speed comparing to Hadoop supported algorithms with the default compression level configured for *Bzip*, *Gzip* and *LZO*.

static look-up table. The slight speed variations are due to the content variation in the source text. In general, AHC is several times faster than *Bzip* and approximately twice as fast as *Gzip*. Compared with *Gzip*, AHC does not use the *LZ77* algorithm and a single Huffman tree is built for the entire dataset rather than two Huffman trees per block. This is the main reason why AHC is faster than *Gzip*.

AHC decompression speed is also stable. Recall that AHC uses hybrid data structures for decoding symbols. In most situations, AHC can achieve constant time for decoding a symbol. The slight difference is due to the sampling and code length variations. If some symbols are missing from samples, their codes will be much longer than expected, especially when their code length is longer than 12 bits, then they must be decoded using a Huffman tree which takes much longer time.

The compression ratio is content dependent. Figure 4.14 shows the compression results from compressing 2GB of data taken from each dataset, in which *Gzip* and *Bizp* performed the best overall, with AHC the best on *mRNA* sequences. This is because the *Gzip* algorithm is based on the *Sliding Window* technique. It replaces any string that has occurred previously by pointers. Unfortunately, the *mRNA* sequence does not present much repetitiveness that can be easily found in a *Sliding Window* and/or the repetitive stings are too short and that makes it worthless to replace them by pointers. The *Bzip* algorithm uses the block based Burrows-Wheeler Transformation to group the same symbols and/or similar symbols together, therefore more repetitiveness can be artificially created. But, auxiliary information about the transformation must be logged which consumes some extra space. In contrast, AHC statically replaces symbols

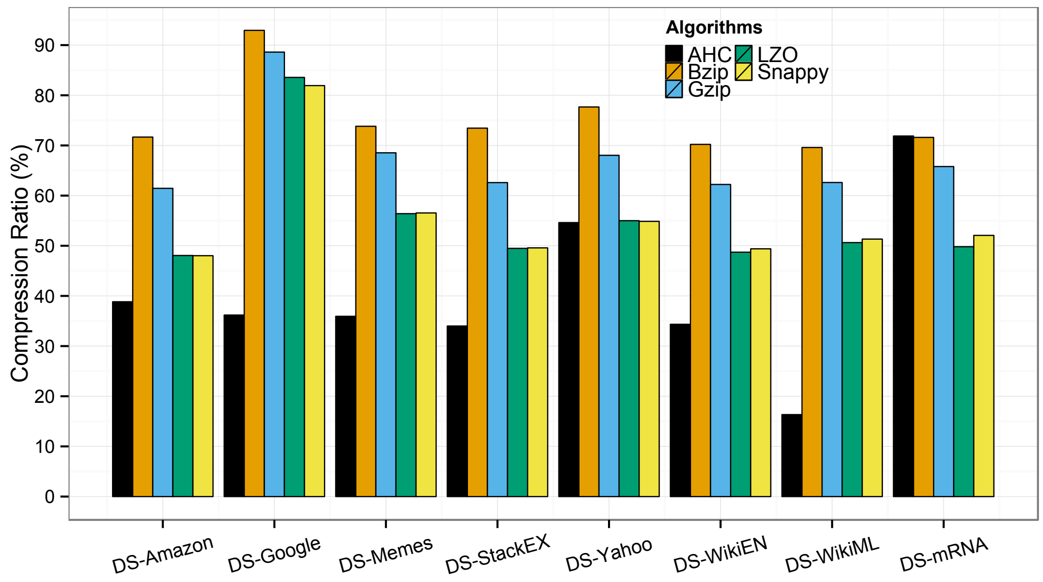


Figure 4.14: AHC compression ratio comparing to Hadoop supported algorithms with the default compression level configured for *Bzip*, *Gzip* and *LZO*.

by codes. The *mRNA* data has very limited symbol richness so that the codewords for symbols are very short. This results in the best compression ratio. In contrast, dataset *DS-WikiML* contains Wikipedia articles in multiple languages. Characters in the standard and extended ASCII table tend to be evenly distributed. This means the dataset has relatively high symbol richness and longer codeword length on average and thus leads to a lower compression ratio.

## 4.3 Conclusion

In this chapter, we introduced the Approximated Huffman Compression scheme. AHC is not a standalone compressor. In fact, it is a solution to the textual data analysis on Hadoop. We have demonstrated the effectiveness of using AHC with Hadoop via two MapReduce jobs with real-world datasets. Evaluation results show that AHC can reduce data size significantly and improve data processing speed accordingly. We use hybrid data structures to improve performance for decoding and pattern searching in AHC compressed data. Theoretical analysis shows an $O(n)$ time complexity. AHC is especially suitable for domain specific and machine generated data. As these types of data often have low symbol richness, AHC can achieve better compression ratios and guarantee to decode in constant time. Additionally, AHC is a character-based encoding scheme. Compared to a word-based scheme, AHC is more flexible in sub-string searching. The main drawback of using AHC with Hadoop is the compatibility issue. Existing algorithms or software packages by default work at the byte level. In order to work with AHC, a translation layer or rewrite of existing software packages seem inevitable.

From another perspective, storing and analysing data in the public environment often raises privacy concerns. Many large volume datasets, for example, cluster usage traces and music ratings, may not be worth encrypting, but often some data fields are anonymized using hashes. To retrieve the original information, an additional set of mapping files must be maintained. In AHC, the compression model can be used as a key to unlock the compressed contents for trusted users. Regarding the security level, it has been shown that Huffman coded data can be difficult to decipher [GMR96]. Considering that we calculate probability distributions for symbols from randomly selected samples, this makes it more difficult for cryptanalysis.

# Chapter 5

# Record-aware Compression (RaC)

*"Those who cannot change their minds cannot change anything."*
— George Bernard Shaw

In the previous chapters, we have seen that context-free compression works out favourably for MapReduce. The context-free aspect makes processing and modifying compressed data in MapReduce feasible. From the experimental results, we have observed that the increase in performance was due to a combination of loading less data from persistent storage to memory; transmitting less data over the network; and processing shorter strings in compressed format. The main limitation with CaPC and AHC is the moderate compression ratio. This can be improved by employing higher-order context-dependent schemes. Currently, several such compression methods have been made available to the Hadoop platform including *Gzip*, *Bzip*, *LZO*, *Snappy* and recently *LZ4*. The main objectives of using these compression methods in Hadoop are to efficiently deliver the intermediate data generated by the Map processes to the corresponding Reducers during the Shuffling phase and to accelerate the process of materializing in-memory data to local persistent storage. In this chapter, we extend the use of these modern compression methods to the data loading phase of MapReduce, while ensuring the compressed data is splittable to HDFS and the split data can be decompressed by Mappers independently in a distributed environment.

Using compressed data (compressed by modern compressors) raises a concern on MapReduce parallelism. Recall that HDFS organizes big files by splitting them into a series of data blocks. Data blocks are distributed across computational nodes. In the MapReduce context, each data block is assigned to a dedicated Mapper and processed independently. This poses a question as to whether each data block can be decompressed independently. In other words, whether the entire compressed data is splittable. Modern compressors endeavour to achieve higher compression ratios by leveraging contextual data and meta-information about the data. This creates dependencies between compressed data and forces the decompression to be a sequential process starting at the beginning of the compressed data. Splitting compressed data at arbitrary points makes data blocks highly unlikely to be decompressable except for the first data block. In order to allow a MapReduce program work with compressed data, all HDFS data blocks must be assigned to a single Mapper due to the sequential decompression process. This defeats the data parallelism goal and is inappropriate for the MapReduce computational model. Additionally, the decompression process takes non-negligible time, as for example when decompressing several gigabytes or terabytes of compressed data sequentially.

In order to address the splittable issue, hope for a solution is given by block-based compression. In general, a block-based compressor takes a block of input data and produces variable-length compressed output. At the end of the each compression block, the *compression context buffer* [1] needs to be flushed fully to make the current compressed data block self-contained. In addition, the boundaries of each compressed data block must be logged, since they are variable-length. In principle, any block-based compression can be made splittable. However, frequently flushing the compression context buffer greatly impairs compression ratio [Sal08] [WMB99]. The *LZO*-splittable algorithm [Twi] has been made available to Hadoop by Twitter. Twitter uses the standard *LZO* algorithm to compress data, then uses an extra indexing program to determine and record splittable boundaries in a separate file. But, this is not the end of the story. Recall that each HDFS data block is assigned to a dedicated Mapper and the Mapper process will be launched on the cluster node where the HDFS data block resides, hence moving the computation. Moreover, each HDFS data block will be further organized as a logical data *split* in the context of the MapReduce framework. A data split guarantees no partial records at the boundaries of a HDFS data block. A data split is further divided into logical records which are often specific to the dataset and the MapReduce programs. The *Record Reader* component of MapReduce is responsible for supplying records to Mappers and a Mapper will process a single record at a time. Therefore, the MapReduce paradigm can be also seen as record-oriented processing.

It is very likely that each HDFS data block will contain incomplete records at the boundaries. Using original data, if a HDFS data block contains a partial record at the beginning of the block, the partial record will be ignored by the MapReduce *Record Reader*; if a partial record is at the end of the HDFS data block, the remaining part of the partial record will be streamed from the immediate succeeding HDFS data block to the current HDFS data block, to make a complete record. Note that if the immediate succeeding HDFS data block is located on the same cluster node, streaming partial records can be done quickly. More often, HDFS data blocks are distributed across cluster nodes,

[1] In many compression algorithms [ZL78] [ZL77] [Wel84], an internal memory buffer is often used to cache a part of the previously compressed input stream as a dictionary for encoding successive data in the stream. This is often the place where the dependencies are created.

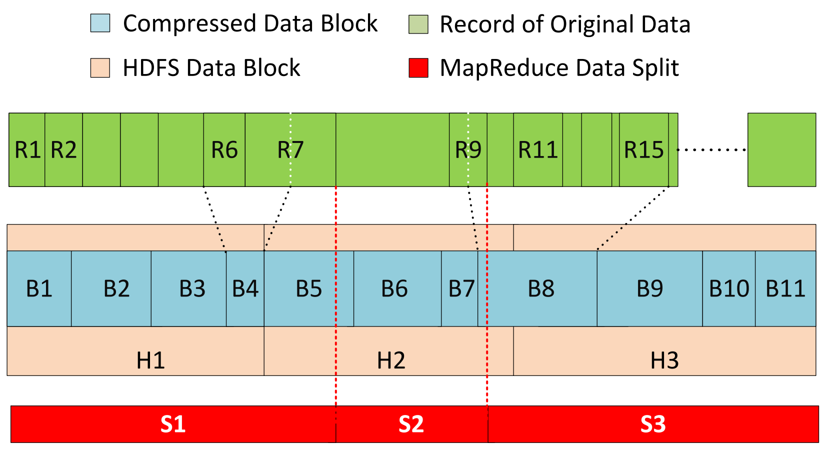


Figure 5.1: An illustration of the issues with none content-aware compression in HDFS.

and data must be streamed across the network which takes much longer time. To this end, a data split is formed and ready for processing. This can become complicated when forming a data split from compressed data. Figure 5.1 shows an example of a blind compression (compression without being aware of record completeness and block boundaries). There are two extreme scenarios.

- In the first scenario, the original record *R6* and the first part of *R7* are compressed in *B4*. The compressed data blocks from *B1* to *B4* are allocated to *H1*. *B5* to *B7* and part of *B8* are assigned to *H2*. (Note that in a real application scenario, the splitting point is purely determined by the pre-defined HDFS block size for a Hadoop cluster.). Assume that three Mappers are started simultaneously. They work concurrently and each Mapper is assigned to a HDFS data block in ordinal order. Before *Mapper-1* starts processing the last record (*R7*) in *B4* of *H1*, the MapReduce *Record Reader* that serves *Mapper-1* must ensure that the last record (*R7*) is a complete record. If the record is incomplete, the *Record Reader* component must stream the second part of the record from *H2* (the immediate succeeding HDFS data block of *H1*). Since data is compressed, in order to stream the second part of *R7*, the entire block of B5 must be streamed to *H1*. Then *B5* is decompressed in order to figure out the second part of *R7*. This can be a time consuming process especially when the size of *B5* is relatively large and consequently delays the completion time of the entire Map processing phase.

  At the same time, *Mapper-2* decompresses *B5*, skips the first record (which is the second part of *R7*) and then starts from the second record.

- In the second scenario, when a compressed data block is split into two different HDFS data blocks as shown in Figure 5.1 for *B8*, this can be more problematic as the first part of *B8* is not decompressable (it will fail the data integrity check, for example, using Cyclic Redundancy Check (CRC) algorithm [PB61]). The *Record Reader* that serves *Mapper-2* must be notified beforehand with the assistance of an extra logging file(s). Thus, before *Mapper-2* starts decompressing the first part of *B8* in *H2*, the remaining part of *B8* in *H3* must be streamed to *H2*. Furthermore, when *Mapper-2* starts processing the last record in *B8*, it still needs to stream block *B9* of *H3* to *H2* for verifying record completeness. At the same time, *Mapper-3* must be notified that the second part of *B8* must be ignored and it should start from *B9*.

Another side-effect is when the compressed data blocks (for example *B8* and *B9*) are relativity large, frequently streaming data blocks between computational nodes can break the data-locality property which is one of the design principles of MapReduce, namely *moving computation to data*. Also, because of this data streaming process between HDFS data blocks, the logical data splits which are actually assigned to Mappers for processing, have different sizes as shown in Figure 5.1 (*S1*, *S2*, *S3*). The different data split sizes leads to different completion time among Mappers, which will consequently delay the overall completion time of the MapReduce program.

In this context, we introduce Record-aware Compression (RaC). RaC is a block-based compression. It takes variable-length input and produces fixed-length compressed output. Each compressed block is guaranteed to contain a set of complete records. A record is determined by user defined rules expressed in *Regular Expression* [2] clauses. Because of the fixed-length blocks, the compressed data can be easily split and MapReduce *Record Readers* do not have to track the block indices, hence it is splittable and lightweight. More importantly, there is no need for a decompression process to determine the logical record completeness at the boundary of each HDFS data block as explained above.

## 5.1 Compression Scheme

In a conventional MapReduce program, the process starts with the reading of blocks from HDFS. The *Record Reader* component is responsible for parsing and splitting the data block into records and then supplying a single record to the corresponding Mapper for processing. While the Mapper is doing the processing, the *Record Reader* starts preparing the next record for the Mapper in the background. When the Mapper completes processing the current record it does not have to wait for the *Record Reader* to parse the next record, thus improving performance. Mappers will generate intermedi-

[2]"*Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings*" [The13].

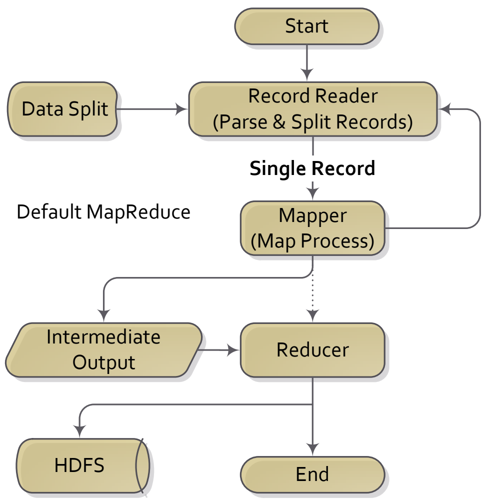


Figure 5.2: An illustration of the default work-flow of MapReduce.

ate data and the intermediate data will be distributed to corresponding Reducers over the network for further processing. The final results produced by Reducers will be eventually stored in HDFS. This work-flow is illustrated in Figure 5.2.

Because this interaction between a Mapper and its associated *Record Reader* influences MapReduce performance, three strategies have been designed for using RaC with MapReduce as illustrated in Figure 5.3.

1. In *strategy-1*, the *RaC Record Reader* [3] is only responsible for reading a fixed-size block of data (RaC-Block) as shown in Figure 5.3 (*left*, Label 1). In this case, the *RaC Record Reader* is lightweight as it only reads one RaC-Block at a time and supplies the RaC-Block to the corresponding Mapper. The Mapper is responsible for decompressing the received RaC-Block, then parsing and splitting records as shown in Figure 5.3 (*left*, Label 2). This is useful when the RaC-Block is relatively large and the computational power of the Mapper is relatively high, so that the difference between the time spent on loading the RaC-Block by the *RaC Record Reader* and processing the RaC-Block by the Mapper is small. Thus the waiting time, as discussed above, can be minimized.

2. In *strategy-2*, the *RaC Record Reader* reads a RaC-Block and decompress it. The decompressed results are then fetched to the corresponding Mapper. In fact, the *RaC Record Reader* supplies a group of records to the Mapper as shown in Figure 5.3 (*middle*, Label 1). The Mapper is responsible for parsing and splitting records as shown in Figure 5.3 (*middle*, Label 2). This strategy is designed to deal with dataset(s) containing small records. For example, the *Yahoo! Music*

[3] By default, the MapReduce framework provides a set of pre-defined *Record Readers* to deal with various data formats. In the RaC scheme, we provide a set of specially designed *Record Readers* (*RaC Record Reader*) to deal with RaC compressed data.

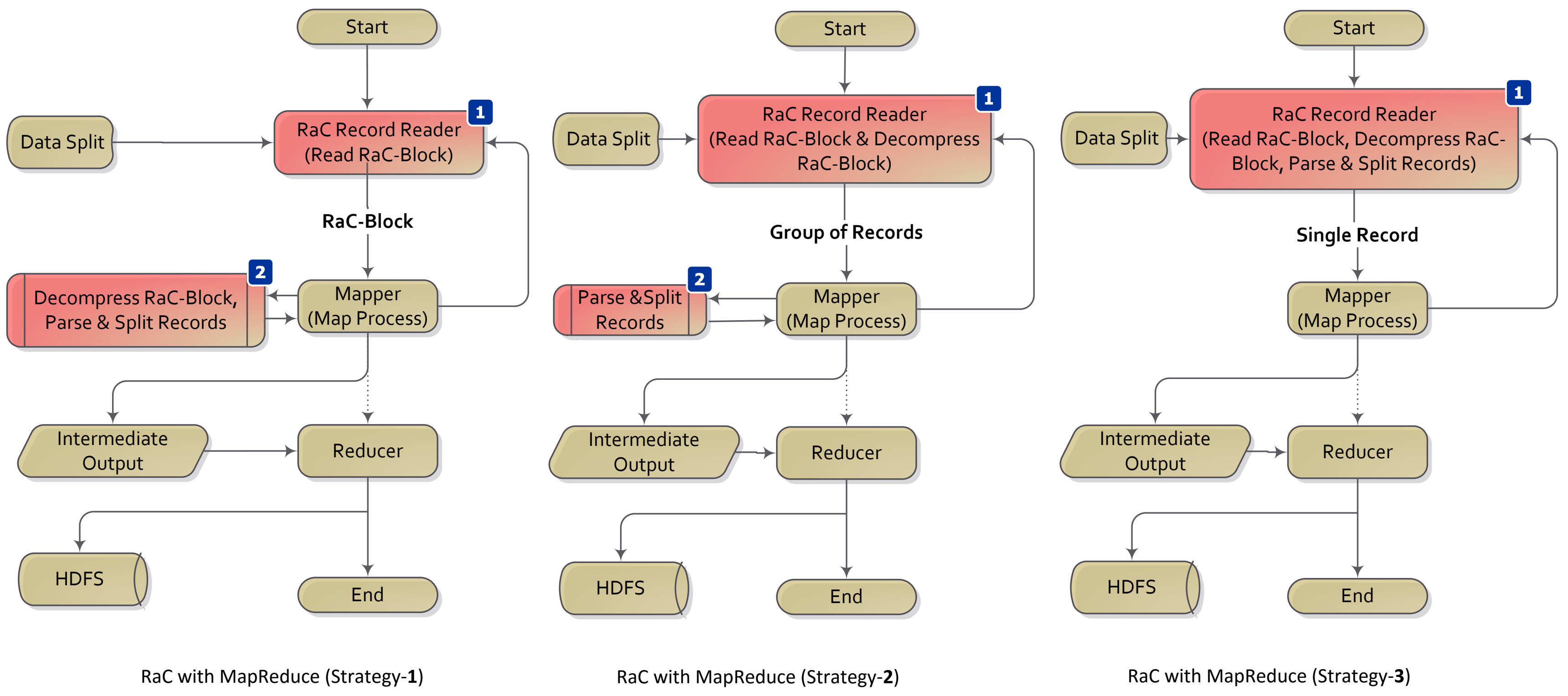

Start
RaC Record Reader (Read RaC-Block)
1
Data Split
RaC-Block
Decompress RaC-Block, Parse & Split Records
2
Mapper (Map Process)
Intermediate Output
Reducer
HDFS
End
RaC with MapReduce (Strategy-1)
Start
RaC Record Reader (Read RaC-Block & Decompress RaC-Block)
1
Data Split
Group of Records
Parse &Split Records
2
Mapper (Map Process)
Intermediate Output
Reducer
HDFS
End
RaC with MapReduce (Strategy-2)
Start
RaC Record Reader (Read RaC-Block, Decompress RaC-Block, Parse & Split Records)
1
Data Split
Single Record
Mapper (Map Process)
Intermediate Output
Reducer
HDFS
End
RaC with MapReduce (Strategy-3)

Figure 5.3: Three strategies designed for using RaC compressed data with MapReduce programs.

*Rating* dataset (*DS-Yahoo*) contains approximately 700 million records. The average record length is about 16 characters long. Supplying a single record to a Mapper for processing as done by conventional MapReduce programs is extremely inefficient. In contrast, the *RaC Record Reader* supplies a group of records to a Mapper at a time, so that we can reduced the unnecessary burden on the *RaC Record Reader* component and at the same time improve the memory utilization at Mappers.

3. In *strategy-3*, the *RaC Record Reader* is responsible for reading, decompressing, parsing and splitting records. A single record is therefore fetched to the corresponding Mapper for processing as shown in Figure 5.3 (*right*, Label 1). This strategy is designed for dataset(s) containing large records where processing a record will take a long time. Also, this strategy makes the implementation of a Mapper generic, as the Mapper does not have to contain the record parsing and splitting logic.

Apart from the RaC usage strategy, RaC compression has two implementations which are based on the *Deflate* (RaC-Deflate) and *LZ4* (RaC-LZ4) algorithms respectively, where RaC-Deflate offers high compression ratios and RaC-LZ4 is speed-optimized. The reason for implementing RaC in two flavours is because the compression ratio and decompression speed often conflict. When using RaC with MapReduce we can save time by loading less data from persistent storage to memory, but at the same time, we must spend extra time on decompressing the data. We aim to identify whether the higher compression ratio with relatively slower decompression speed approach or the lower compression ratio with higher decompression speed approach can achieve better overall performance.

Recall that RaC takes variable-length input and then produces fixed-size compressed output. We use a 512KB block size for RaC-Block (the size is adjustable). This makes every 512KB of compressed data self-contained. Using a naïve approach, we can fetch a single record at a time to the RaC compressor until a RaC-Block is full, then we start from a new RaC-Block. In practice, parsing records can be very time consuming particularly when a complicated delimiter(s) is used to split records. To improve compression speed, RaC starts with a lightweight sampling process (RaC uses the same sampling strategy explained in Section 3.1.3). Each sample drawn from the underlying dataset is a fixed-size block (Sample-Block) of data. We use a default 128KB Sample-Block size. Each sample is compressed using *Deflate* or *LZ4* (depending on whether RaC-Deflate or Rac-LZ4 are used) and the mean length of the compressed samples is used to estimate how much data will be needed to fill a 512KB RaC-Block. In other words, the estimation results are used to decide the input buffer size for RaC compression.

Once the input buffer size is calculated, a block of data will be read into the input buffer. Starting from the end of the input buffer, we seek the first record delimiter

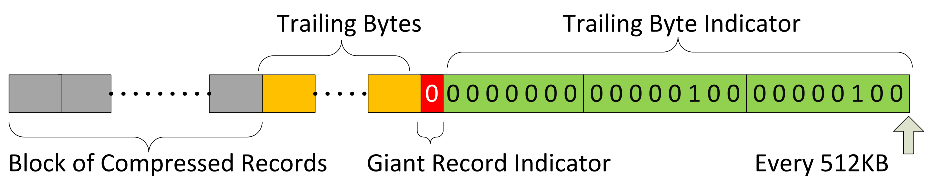


Figure 5.4: RaC block-packaging scheme.

defined by the user. Any partial record will be removed from the current input buffer and postponed to the next iteration. This ensures that the input buffer contains a set of complete records and so the corresponding compressed data block is self-contained. After compressing an input block, the size of the compressed data block must be less than or equal to the size of the RaC-Block. Since we guarantee the same RaC-Block size, any gaps will be filled up by trailing bytes. The last three bytes of the RaC-Block are reserved and used to indicate how many trailing bytes have been appended. Additionally, the Most Significant Bit (MSB) of the third last byte is used to indicate a giant record which spans multiple RaC-Blocks. The packaging format is illustrated in Figure 5.4. In the situation where the compressed data size is greater than the size of the RaC-Block, a single full record will be removed from the current buffer (at the end of the buffer) and postponed to the next iteration. The process continues until the size of compressed data is less than or equal to the size of RaC-Block with the consideration of the three reserved trailing bytes. This also implies that the accuracy of the estimation of the input buffer size from sampling is an important factor affecting RaC compression performance.

## 5.2 Evaluation

We evaluate the effectiveness of RaC using an on-site Hadoop cluster. The cluster topology and configuration are given in Appendix A (Figure A.1 and Table A.1). The experimental environment was configured with Hadoop version 2.5.0 and Linux kernel version 2.6.32. The Hadoop HDFS was configured with a single replication and 64MB block size. A set of standard MapReduce jobs were used for evaluating RaC with MapReduce as summarized in Table 5.1. The RaC strategy-2 was used across all experiments. The evaluation was conducted using a collection of real-world datasets as listed in Appendix A (Table A.2). The evaluation includes MapReduce performance, cluster storage requirements, memory constraints, RaC compression ratio and RaC compression/decompression speed.

Table 5.1: A summary of MapReduce jobs used for evaluating the RaC compression scheme.

| Job | Dataset | Data Type | Input | Intermediate | Output | Memory Allocation | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|---|
| Sentiment Analysis | DS-Amazon | O : | 33.4 GB | 834.5 MB | 990.0 MB | 12.0 GB | 23m35s | | |
| | | D : | 12.5 GB | 798.8 MB | 990.0 MB | 7.8 GB | 15m09s | 35.8% | 62.6% |
| | | L : | 20.5 GB | 812.8 MB | 990.0 MB | 9.3 GB | 18m17s | 22.6% | 38.6% |
| Page Rank | DS-Memes* | O : | 50.5 GB | 630.5 MB | 19.1 MB | 10.2 GB | 21m17s | | |
| | | D : | 28.1 GB | 630.5 MB | 19.1 MB | 4.2 GB | 15m29s | 27.3% | 44.4% |
| | | L : | 40.3 GB | 630.5 MB | 19.1 MB | 8.3 GB | 19m10s | 9.9% | 20.2% |
| Server Load Analysis | DS-Google | O : | 158.9 GB | 530.9 MB | 371.1 KB | 21.2 GB | 42m10s | | |
| | | D : | 46.9 GB | 315.4 MB | 371.1 KB | 14.8 GB | 29m05s | 31.0% | 70.5% |
| | | L : | 79.1 GB | 530.8 MB | 371.1 KB | 19.3 GB | 38m32s | 8.6% | 50.2% |
| 5-Gram | DS-WikiEN* | O : | 13.0 GB | – | 63.1 GB | 3.9 GB | 7m50s | | |
| | | D : | 3.2 GB | – | 63.1 GB | 2.5 GB | 5m00s | 36.2% | 75.4% |
| | | L : | 3.9 GB | – | 63.1 GB | 2.6 GB | 5m21s | 31.9% | 70.0% |

*continued . . .*

*. . . continued*

| Job | Dataset | Input Type | Input | Intermediate | Output | Memory Allocation | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|---|
| Word Count | DS-StackEX | O : | 40.6 GB | 20.9 GB | 9.0 GB | 22.9 GB | 44m57s | | |
| | | D : | 19.3 GB | 13.1 GB | 9.0 GB | 19.1 GB | 40m23s | 10.2% | 52.5% |
| | | L : | 20.3 GB | 13.1 GB | 9.0 GB | 20.4 GB | 38m23s | 14.6% | 50.0% |
| Music Rank | DS-Yahoo | O : | 10.2 GB | 3.0 GB | 3.2 MB | 5.0 GB | 11m50s | | |
| | | D : | 3.3 GB | 3.0 GB | 3.2 MB | 3.9 GB | 9m30s | 19.7% | 67.7% |
| | | L : | 5.7 GB | 3.0 GB | 3.2 MB | 4.5 GB | 10m06s | 15.2% | 43.3% |
| Finding Implant Sequence | DS-mRNA | O : | 12.5 GB | 21.5 MB | 17.0 Bytes | 2.0 GB | 4m22s | | |
| | | D : | 5.3 GB | 9.6 MB | 17.0 Bytes | 951.7 MB | 2m03s | 53.1% | 57.6% |
| | | L : | 10.8 GB | 18.8 MB | 17.0 Bytes | 1.5 GB | 3m18s | 24.4% | 13.6% |

O: Original datasets
D: RaC-Deflate compressed datasets
L: RaC-LZ4 compressed datasets
*1GB = 1,073,741,824 Bytes*
*: Subset of the data was used for the evaluation.

### 5.2.1 Performance

Table 5.1 contains our main experimental results. We use seven standard MapReduce jobs to evaluate the effectiveness of RaC with MapReduce. We run each job three times with the original, RaC-Deflate and RaC-LZ4 compressed data, respectively. On average, RaC-Deflate and RaC-LZ4 can reduce data size by 61.5% and 40.8%; and improve analysis performance by 30.5% and 18.2%.

In the *Sentiment Analysis* job, we use the *SentiWordNet* lexicons [ES06] to mine customer opinions on Amazon products. In comparison to the *Sentiment Analysis* carried out in the CaPC evaluation (Chapter 3, Section 3.2), we use the full dataset which contains approximately 35 million reviews for $\sim$2.5 million products given by $\sim$6.6 million Amazon online store customers. Using RaC-Deflate and RaC-LZ4 compression, the original data size is reduced by 62.6% and 38.6%, respectively. As a result, when using RaC compressed data, we can save a significant amount of time for loading data from persistent storage to memory as indicated by the performance gains of 35.8% and 22.6%.

The *Server Log Analysis* was selected to evaluate RaC with a comparatively large analysis task. The main task is to determine the over/under utilized servers from Google cluster log files. For this particular dataset, RaC-Deflate and RaC-LZ4 reduce the original data size by 70.5% and 50.2%, respectively. The performance gains of 31% and 8.6% mainly come from loading less data (RaC compressed) from disk to memory and distributing smaller intermediate output to Reducer nodes over the network. The different size of the intermediate output is a result of the MapReduce *Local Combiner* effects [4]. In this evaluation, the MapReduce *Local Combiner* was used for the *Sentiment Analysis*, *Server Log Analysis* and *Finding Implant Sequence* jobs. The details about the *Local Combiner* effect will be explained in Chapter 6 (Section 6.3).

Recall that a standard MapReduce job generally consists of Map tasks and Reduce tasks. The Map phase comprises of *Record Reader*, *Map Logic*, *Partition*, *Sorting*, *Spilling*, *Merging* and *Shuffling* steps. The Reduce tasks can be further divided into *Sorting*, *Merging*, *Reduce Logic* and *Spilling* steps. The *5-Gram* job consists of Map tasks only. There are no *Partition*, *Sorting* and *Shuffling* phases, thus the network I/O is kept to a minimum. In the experiment, the observed performance gains come purely from the data loading processes. As shown in Figure 5.5 (*5-Gram*), the aggregated CPU times spent on the Map tasks are similar for the analysis with the original, RaC-Deflate and RaC-LZ4 compressed data (as indicated by *Map_Task_CPU_Total*). In contrast, the aggregated total time spent on Map tasks differ largely, as indicated by *Map_Task_Total*. Subtracting the time spent on the *Map Logic* (indicated by *Map_Task_CPU_Total*) from the total time spent on the Mapper (indicated by *Map_Task_Total*), the difference is mainly due to

[4]MapReduce uses a combining function to merge records having the same key in the output of a Mapper.

loading different size data from disk to memory.

In the *Word Count* analysis, the job with RaC-LZ4 compressed data performed better than the job with RaC-Deflate compressed data. This is because both RaC-Deflate and RaC-LZ4 achieve a similar compression ratio (52.5% and 50.0%) and both jobs produce the same size of intermediate output. The only major difference is the decompression speed. Recall that the *RaC Record Reader* reads a block of data and decompresses it in memory, then forwards the decompressed data to the corresponding Mapper. As shown in Figure 5.9 and Figure 5.10, the decompression speed of RaC-LZ4 is approximately 20% faster than the RaC-Deflate performed on the *StackExchange Posts* dataset (*DS-StackEX*). This leads directly to the 14.6% performance gain from using RaC-LZ4 compressed data compared to the 10.2% gained from RaC-Deflate. However, when the data size becomes larger as shown in the *Finding Implanted Sequence* jobs, we can find that the decompression speed becomes much less significant.

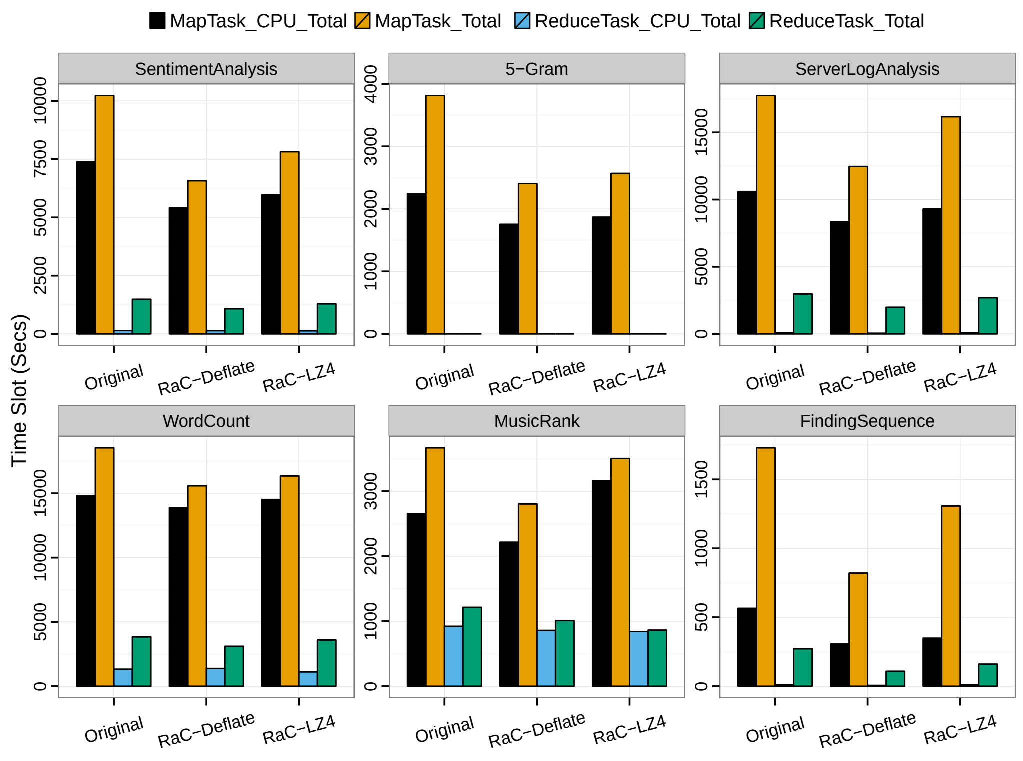


Figure 5.5: Aggregated CPU time and total time consumed by both Map and Reduce tasks.

In the *Music Rank* and the *Page Rank* jobs, the MapReduce *Local Combiner* was disabled. This forces Mappers to produce the same size of intermediate data. Because each Mapper is treated as an independent process, Mappers generate their own output data. The total number of intermediate data blocks produced by Mappers differs largely as indicated by the MapReduce Framework Counter: *Merged_Map_outputs* = $\{815, 265, 460\}$ for {Original, RaC-Deflate, RaC-LZ4} (this is the same as the number of Mappers)

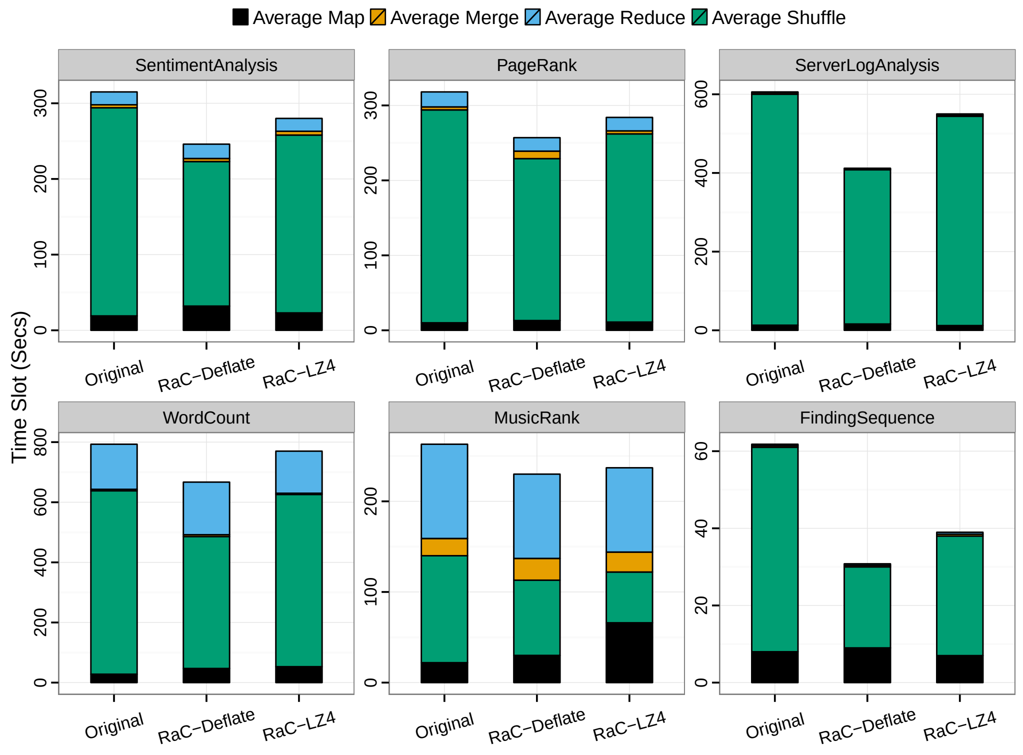


Figure 5.6: Averaged time allocation for MapReduce Map, Shuffle, Merge, and Reduce tasks.

for the *Music Rank* job. Recall that the Map output data needs to be sorted and partitioned before the *Shuffling* phase. And during the *Shuffling* phase, Map output data must be distributed to the designated Reducers according to the *Partition* results. The higher volume of Map output data potentially leads to a higher number of partitions. Distributing highly fragmented data from Map nodes to Reducer nodes will increase the transmission delay on the network due to the fact that each transmission requires a separate network connection. It may also create disk I/O contention when reading multiple data blocks from the same disk concurrently for parallel data transmission. It also increases the delay on Reducer nodes because the partitioned data that has been received from Mappers needs to be sorted again and then merged. This is shown in Figure 5.5 as indicated by the difference in the *Reduce_Task_Total* and in Figure 5.6 as indicated by the large variation in the *Average Shuffling* time.

In addition to the *Music Rank* job, the *Yahoo! Music Rating* dataset contains a large number of small records (approximately 16 characters per record on average) and processing effort for each record is lightweight. Using the original dataset, the MapReduce *Record Reader* is heavily loaded due to supplying a single record to a Mapper at a time. Using RaC compressed data, we can supply a block of records to a Mapper which makes the *Record Reading* process more efficient.

In summary, we observe a considerable improvement in analysis performance for all types of jobs and a significant data size reduction. RaC shows its particular usefulness on reducing the burden on the MapReduce *Record Reader* component and enhancing the effects of the MapReduce *Local Combiner*. We also find that the decompression overhead does not have much influence on analysis performance. Instead, the size of the input data and the intermediate output data play vital roles. In general, using RaC-Deflate or RaC-LZ4 compressed data requires much less memory to be allocated. The reasons are as follows. Mappers are independent processes and each Mapper has a dedicated Java Heap Space assigned to it. The higher number of Mappers leads to the more memory being required. Because the source data size is the dominant factor affecting the number of Mappers, with compressed data, the data size has been significantly reduced. This results in fewer Mappers and consequently less aggregate memory consumption. Moreover, supplying a block of data to a Mapper at a time can use system memory more efficiently. This is particularly useful for a multi-tenant cluster in which the saved memory can be assigned to more users.

### 5.2.2 Compression Speed and Ratio

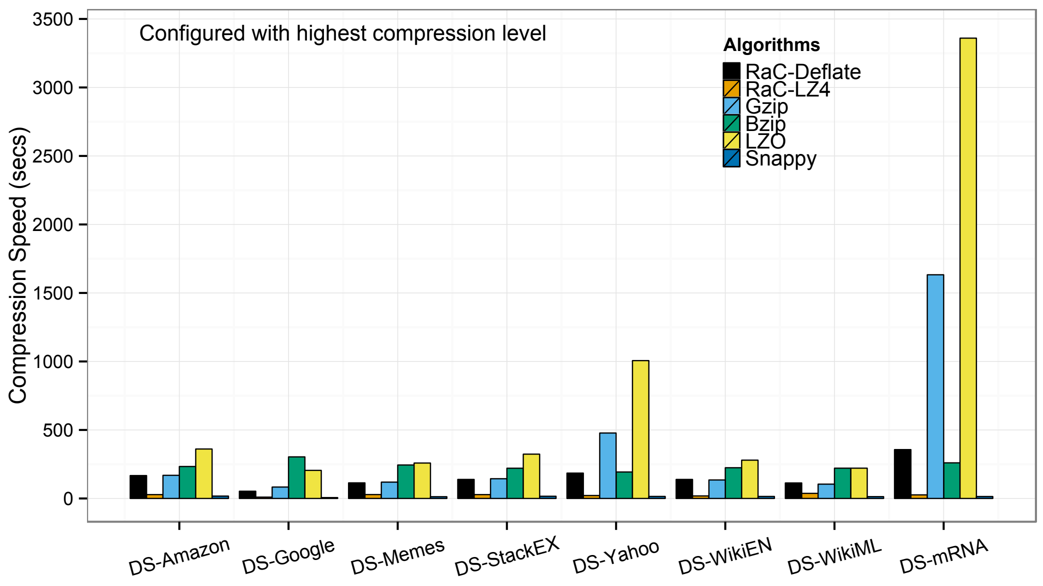


Figure 5.7: RaC compression speed comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

The RaC compression/decompression speed and ratio are compared with *Gzip* (*v1.6*), *Bzip* (*v1.0.6*), *LZO* (*v1.03*) and *Snappy* (*v1.1.2*). The time requirements include the *tar* process. 2GB data drawn from each dataset listed in Appendix A (Table A.2) was used in the evaluation. It should be noted again that the compression speed and ratio are often in conflict. Modern compressors provide flexibility in adjusting the weighting

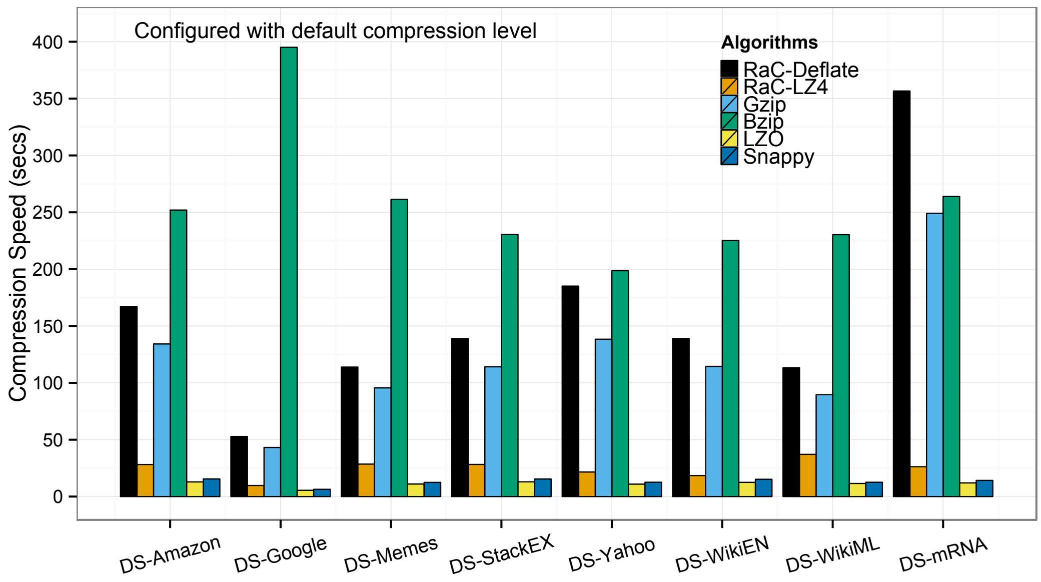


Figure 5.8: RaC compression speed comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

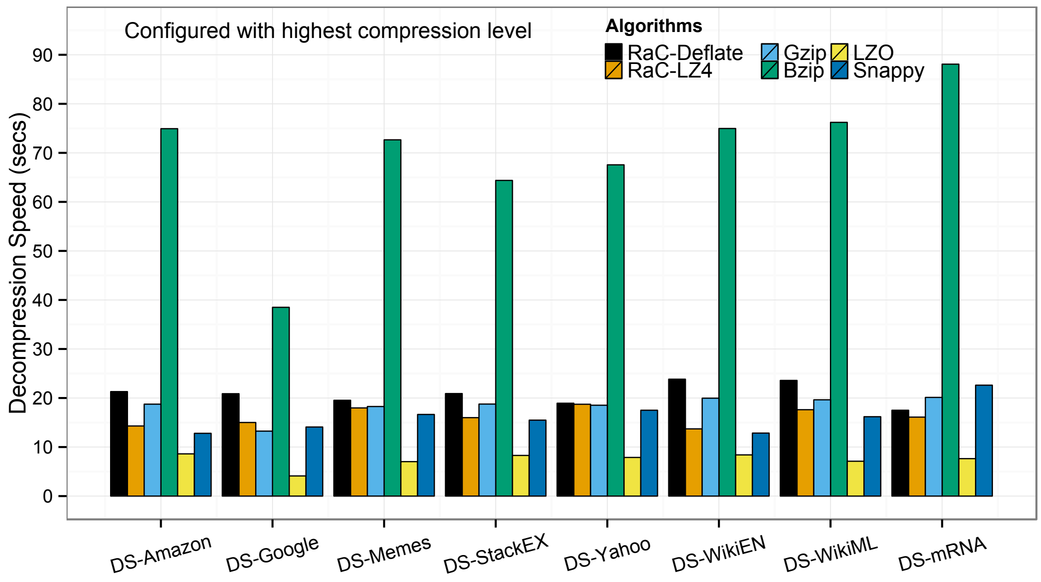


Figure 5.9: RaC decompression speed comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

between compression speed and ratio to meet user requirements. Experiments in this section provide two configurations (highest compression ratio and fastest compression speed) for the aforementioned compression schemes. Figure 5.7 (*configured with fastest compression speed*) and Figure 5.8 (*configured with highest compression ratio*) show that the compression speed of RaC-Deflate is similar to *Gzip* and RaC-LZ4 is close to *Snappy*. The main reason for the slight variations in the compression speed is due to the extra

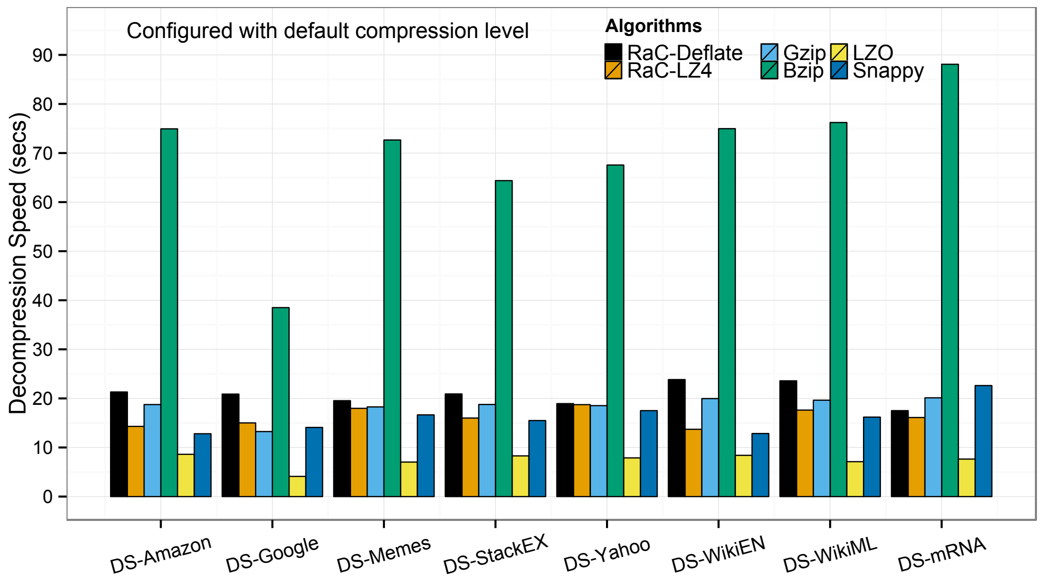


Figure 5.10: RaC decompression speed comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

processes for sampling, estimation of RaC-Block size and determining record completeness a the end of each RaC-Block. Generally, RaC-LZ4 is much faster when compared to *Bzip*, *Gzip* and RaC-Deflate. This is because of the speed-oriented nature of the *LZ4* algorithm. But, comparing with *LZO* and *Snappy*, RaC-LZ4 is still slower due to the same reasons given for RaC-Deflate (the extra processing involved in compression). Note that *LZO* is exceptionally slow when the highest compression level is used. In decompression, RaC-Deflate performs similarly to *Gzip*, and RaC-LZ4 performs similarly to *Snappy*, as shown in Figure 5.9 and Figure 5.10. Interestingly, the configuration of compression level has almost no impact on the decompression speed. This is shown in both Figure 5.9 and Figure 5.10. These two figures are almost identical.

Compression ratio is content dependent. Figure 5.11 and Figure 5.12 show the compression results from compressing 2GB data drawn from each dataset listed in Appendix A (Table A.2). RaC-Deflate and RaC-LZ4 do not have options for configuring compression levels. Thus, the compression ratios are identical as shown in both figures. Because both RaC-Deflate and *Gzip* are based on the *Deflate* algorithm, they performed similarly. In contrast, the compression ratio of RaC-LZ4 varies greatly depending on the data contents. This is because, internally, the *LZ4* algorithm will break a sample data block into smaller data blocks related to the size of the internal buffer used. *LZ4* firstly examines whether the current input data is compressible or not. If the data is not compressible, it will simply skip the current data block without further trying and move to the next data block in order to accelerate the compression process. During the RaC sampling process, if *LZ4* performs badly on the sample data as explained above, the mean length of the compression results will be longer. Eventually the estimated

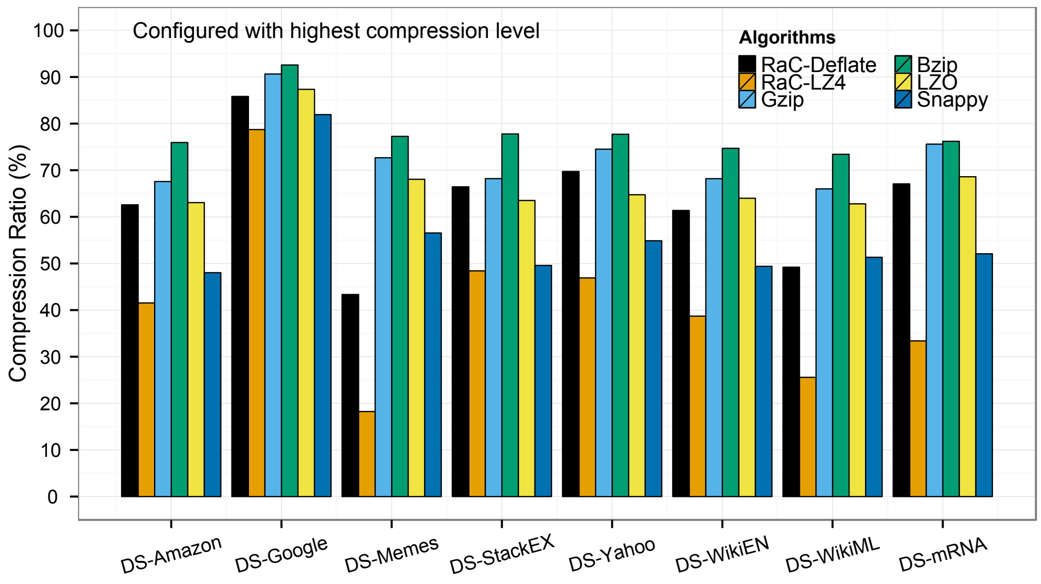


Figure 5.11: RaC compression ratio comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

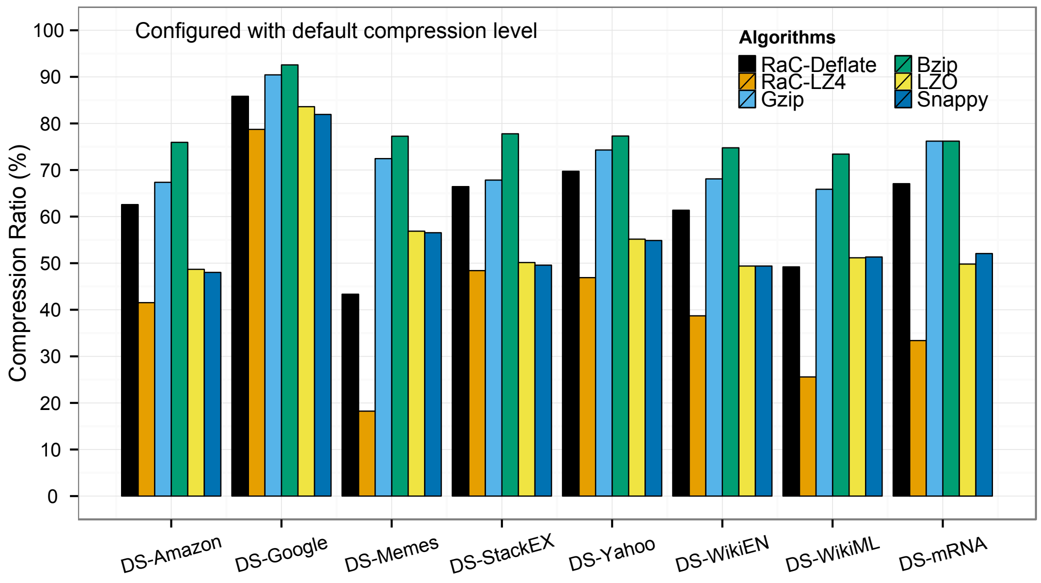


Figure 5.12: RaC compression ratio comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

length for the RaC input buffer will consequently become longer, which leads to poorer compression ratio overall.

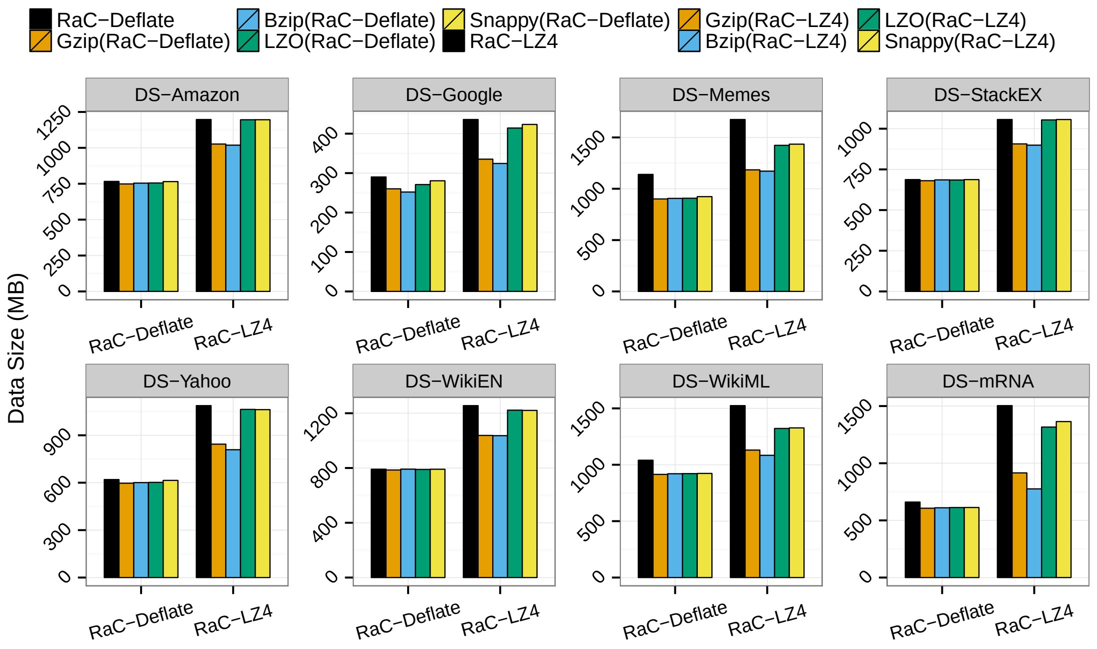


Figure 5.13: Applying Further compression to RaC compressed data.

### 5.2.3 Further Compression

Recall that MapReduce uses compression to reduce the intermediate data so that the cost of distributing the intermediate data over the network can be minimized. Also, data is often compressed for saving storage space in HDFS. In RaC, we employ a relatively loose format for packaging compressed data blocks (Section 5.1). Thus, it may be worth investigating whether applying compression (using the aforementioned standard schemes) to RaC compressed data can further reduce data size so that the data transmission cost can be further reduced and more HDFS storage space can be saved. In this experiment, we take 2GB data drawn from each dataset. We compress the 2GB original data using RaC. The RaC compressed data is then fetched into *Bzip*, *Gzip*, *LZO* and *Snappy*, respectively. We observe that any further compression applied to the RaC-Deflate compressed data does not give much improvement. The slight variation in the RaC-Deflate group (as shown in Figure 5.13) is mainly due to the use of the packaging format in RaC. The compression results indirectly prove that our block size estimation technique is relatively accurate as there are not many trailing bytes used for packaging. In contrast, RaC-LZ4 is a speed-optimized scheme. Applying *Gzip* and *Bzip* to RaC-LZ4 compressed data can further reduce the data size by several hundreds of megabytes. Note that in this experiment, *Gzip*, *Bzip* and *LZO* are configured with their default compression levels.

## 5.3 Conclusion

In this chapter, we introduced Record-aware Compression for Hadoop. In general, RaC can be used with other analysis platforms such as Spark [Apa15] as well as higher level abstractions of MapReduce such as Hive [HCG+14], with extended supporting libraries for each platform. In the evaluation, we show that using RaC can greatly reduce data loading time and the system memory required. More importantly, we observe that the time spent on decompressing data in memory is insignificant compared to the time required for loading data from persistent storage to memory. This can be more important for big data analysis in cloud environments because in cloud environment computational nodes often receive data from remote centralized storage systems rather than from local hard disks. The experimental results lead us to believe that content-aware and data-specific compression is very promising for big data processing and analysis. RaC serves as a preliminary study to the main work presented in the following chapter.

# Chapter 6

# Record-aware Partial Compression (RaPC)

*"The worm in the radish does not think there is anything sweeter."*
— Sholem Aleichem

As previously discussed, the design of the CaPC scheme centres around the separation of informational and functional contents. This is mainly driven by the fact that most Internet generated texts have a format defined by a set of functional characters. Many existing algorithms and software packages rely on those pre-defined characters for processing. CaPC only compresses the informational contents while keeping the functional contents intact. This allows the existing algorithms to work with the CaPC compressed data without modification. The compression is a process of replacing informational contents by specially designed codes which makes the compressed data splittable to the HDFS. More importantly, it can be directly consumed by MapReduce programs.

The main limitation of the CaPC is the limited code space which results in a relatively low compression ratio. In addition, due to the use of a word-based compression method, sub-string searching is not supported. According to our evaluation results, as shown in Section 3.2, CaPC can reduce data size by approximately 30%. In order to improve the compression ratio and support arbitrary string searching, we introduced the AHC scheme in Chapter 4. AHC is a bit-oriented, character-based, context-free entropy encoder based on the Huffman algorithm. Given a text $T[1, n]$ drawn from an alphabet set $\Sigma$, AHC can achieve $O(n \cdot H_0(\Sigma) + \left\lceil \frac{n}{b^{AHC}} \right\rceil (max(\ell) + 7))$ (as given in Section 4.1.2) space efficiency. The higher compression ratio of AHC is due to the lower cardinality of the alphabet $\Sigma$. Because of the smaller data size, using AHC compressed data with MapReduce programs can further reduce the I/O cost when: loading data from persistent storage to memory; materializing in-memory data to HDFS; and transmitting data over the network between cluster nodes. Furthermore, AHC uses a hybrid data structure for decoding symbols in constant time. However, manipulating data at bit level exhibits inefficiencies due to the underlying operating systems and compilers. This can be improved by employing indexing techniques such as *Ranking* and *Selection* [Jac92]. On the other hand, this increases the data size. In contrast, manipulating strings at byte level is much more efficient.

Both CaPC and AHC are context-free coding schemes. In order to achieve higher compression ratios, we must employ a higher order coding scheme or utilize context information during compression. Because the standard compressors such as *Bzip*, *Gzip*, *LZO*, *Snappy* and *LZ4* that are currently supported by Hadoop only allow decompression in a sequential manner, we introduced the RaC scheme in Chapter 5. RaC supports decompression on a per pre-defined block basis. The main goal of RaC is to reduce data size to close to that achieved by general purpose compressors, while at the same time making the compressed data splittable to the HDFS. Thus, the compressed blocks can be supplied to Mappers and decompressed in memory in parallel.

For CaPC and RaC, we have identified that the performance gains in the evaluation came from three sources. The first source is that loading the compressed data, which is much smaller than the original data size, from hard disk to memory takes much less time. The extra time spent on the data decompression in memory is insignificant compared to the data loading time. Especially, when there are multiple Map processes running concurrently on a single physical cluster node, loading smaller data can potentially reduce disk I/O contention. This is the major contribution of the RaC compression. Secondly, when using CaPC, the compressed data is directly consumable by MapReduce programs, and the size of the data flowing in the cluster is smaller than the same analysis using the original text, thus reducing the data transmission cost over the network. When using the RaC compressed data with the MapReduce *local combiner* mechanism, the Map processes produce smaller intermediate output data. This also reduces the data transmission cost during the MapReduce *Shuffling* phase. We explain the third part using an example scenario. Most data-centric analysis involves pattern searching. When searching a pattern of length $m$ from a source data of length $n$, a naïve search algorithm requires $\Theta(m(n-m))$ time; or using the more efficient Knuth–Morris–Pratt algorithm [KJHMP77] requires $\Theta(m)$ preprocessing time and $\Theta(n)$ matching time. In contrast, assuming that CaPC can reduce data size by $\sim$30%, the length of the CaPC compressed source is therefore given by $0.7 \cdot n$ and the compressed pattern length is $0.7 \cdot m$, and this gives $\sim$30% faster searching speed regardless of which algorithm is used. Theoretically, if we combine the two features from the CaPC and the RaC, we can further improve the performance of MapReduce.

To this end, we introduce the Record-aware Partial Compression scheme [DH15b]. RaPC comprises a two-layer compression in which the Layer-1 scheme inherits the main features of CaPC with an improved design and the Layer-2 scheme inherits the main features of RaC with improved compression ratio and speed.

## 6.1 Compression Scheme

The RaPC is a nested two-layer compression scheme. The algorithm used at each layer is specifically designed for improving MapReduce performance and reducing Hadoop cluster resource usage. Each layer is appropriate for the corresponding stage of data processing. This is the novelty of the RaPC scheme. The design goals for the outer layer are to maximally reduce the data size for reducing data loading time while making the compressed data splittable to the HDFS. It is based on a modified *Deflate* algorithm. The inner layer is a word-based context-free compression scheme which makes the compression results directly consumable by MapReduce programs using the supporting libraries.

### 6.1.1 RaPC Layer-1 Compression Scheme

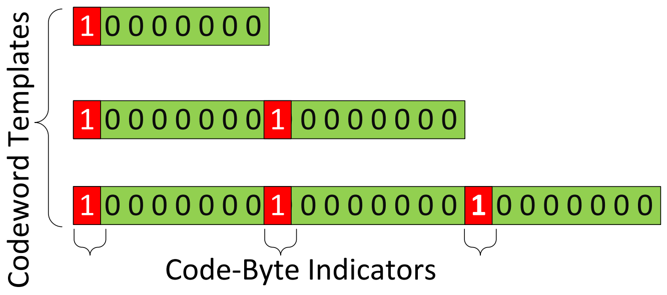


Figure 6.1: RaPC Layer-1 encoding template. The length of codewords increases from one byte to three bytes depending on the amount of compressible information in a given dataset.

The RaPC Layer-1 (RaPC-L1) encoding is a byte-oriented, word-based partial compression scheme. RaPC-L1 separates informational contents and functional contents. Any character or group of consecutive characters from the range [a - z], [A - Z] and [0 - 9] are considered to be informational, other characters are treated as functional. RaPC-L1 only compresses informational contents.

The RaPC-L1 code length grows from one byte to the maximum of three bytes depending on the number of compressible strings in the text. A compressible string is a string that is firstly categorized as informational content. Secondly, the string length must be longer than the current code-length (code-length grows dynamically during the compression). Functional characters are used as delimiters to split informational contents. Every unique compressible string will have an integer value assigned to it in the order of their discovery. The integer value is then used to fill in one of the templates as shown in Figure 6.1. The Most Significant Bit (MSB) of each code-byte is set to one. This ensures that the code-bytes are distinguishable from uncompressed contents. For example, given a message "*Big Data Analysis*", the encoded message is "*10000000*00100000*10000001*00100000*10000010*". During the compression, the three strings are discovered in order. Their corresponding codes are therefore the integer values *0*, *1*, *2*. The integer values will be used to fill in the template as illustrated in Figure 6.1 and subsequently replace the corresponding strings in the compressed message. The byte "*00100000*" is the *white-space* character defined in the standard ASCII scheme. It is not RaPC-L1 compressed but used as a delimiter to split strings. The asterisk symbols are only used to indicate byte boundaries for clear presentation in this example. In addition, Unicode characters often use the extended ASCII codes which have the MSB set to one. In order to avoid conflict, Unicode characters are enclosed with a pair of special characters (*0x11* and *0x12*). *0x11* and *0x12* correspond to the standard ASCII code *Device Control 1* and *Device Control 2* characters, respectively. They are never used in text data, therefore it is safe to use them as special characters for RaPC-L1 compression.

Currently, we do not compress complex Unicode texts such as Chinese or Korean texts, because these languages do not use explicit delimiter(s) to separate words in a sentence. In order to compress complex Unicode texts, a language specific grammar-based parser or dictionary must be employed. This will make the compression content-specific and slower.

Referring to the code template (Figure 6.1), the RaPC-L1 encoding scheme offers a code space of size $2^{21}$. Before the actual compression process starts, a lightweight sampling process is carried out to gather statistics on the most frequently used words. The top $2^7$ words will then be selected. Each selected word will have an integer value assigned to it. This guarantees that these most frequently used words from a given text have the shortest code of one byte. Increasing the number of selected words and/or the sample size will improve the compression ratio, but the compression speed will be slower due to the fact that the process of sorting words by frequency requires $O(n \cdot log(n))$ time, where $n$ is the number of words observed from the samples to be sorted, the higher sampling rate will increase $n$.

During the compression, compressible strings and their corresponding codes are temporarily stored in a *HashMap* data structure. Thus, searching for existing strings has $\Theta(1)$ complexity on average. The compressible strings will be replaced by their corresponding code-byte(s) and non-compressible strings will be sent to the output intact. This gives the RaPC-L1 compression sub-linear complexity in time.

RaPC-L1 generates two output files: one is the compressed file(s), the other is the compression model. The separation of the compressed data and the model is driven by the MapReduce framework. In a MapReduce program, Map processes are independent of each other (the same rules apply to Reduce processes). For many cases in textual data analysis, the textual contents often need to be converted into other data types (e.g., text to integer values, yes/no to boolean values and strings to dates). This requires transforming the compressed data into its original format, then converting to other data types. Because each Mapper loads and processes its assigned data blocks independently, this requires the model file to be available to all Mappers and/or Reducers when it is needed. Technically, the HDFS *Distributed Cache* is the best place to store the compression model file as explained in Chapter 3. The compression model file is a list of compressible strings collected during compression. The index of each string in the list is determined by its corresponding code converted to an integer value. The RaPC-L1 decompression process reads the compression codes and converts them into integer values, then, based on the values, we can quickly identify their corresponding original strings. Also, consider that data is partially compressed by RaPC-L1, therefore the RaPC-L1 decompression exhibits sub-linear complexity in time. Another point worth noting is that the model file can be reused for compressing future data. If data files are generated from the same source, they often share a common vocabulary, as, for example, data generated by machines.

Recall that the RaPC-L1 scheme compresses data partially. Functional characters such as *white-space*, *comma* and *new-line* are left untouched. This unique feature allows a program to manipulate the compressed data freely, for example, splitting fields by a comma, removing words and adding new contents, etc. Furthermore, it guarantees the manipulated contents are still decodable. However, it should be noted that using the RaPC-L1 compressed data on Hadoop poses a concern. Hadoop currently only supports ISO-8859-1 and UTF-8 encoding for text. Because of the use of the extended ASCII codes in RaPC-L1 compression, the RaPC-L1 compressed contents are no longer in a traditional text format. In another words, the data cannot be simply converted to strings (e.g., UTF-8 encoded format). This is because of the irregular use of the extended ASCII characters (MSB set to 1). Thus, the RaPC-L1 compressed data must be treated as byte sequences and processed as byte sequences.

### 6.1.2 RaPC Layer-2 Compression Scheme

The RaPC Layer-2 (RaPC-L2) compression can be used with the original text or used to compress the RaPC-L1 compressed data. The purpose of RaPC-L2 is to maximally reduce data size and package logical records into fixed-length blocks which then makes the compressed data splittable to the HDFS. It is based on a modified *Deflate* algorithm.

The conventional *Deflate* algorithm is block-based [Deu96]. The actual *Deflate* compression employs a dictionary method (*LZ*77 [ZL77]) and a statistical method (Huff-

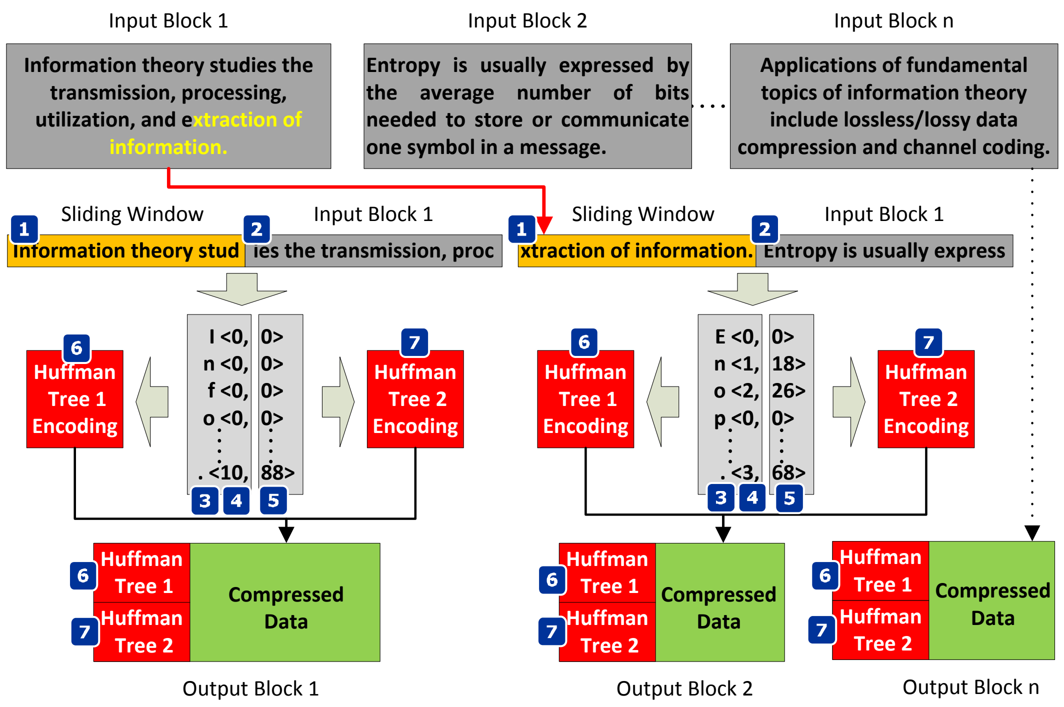

Input Block 1
Information theory studies the transmission, processing, utilization, and extraction of information.
Input Block 2
Entropy is usually expressed by the average number of bits needed to store or communicate one symbol in a message.
Input Block n
Applications of fundamental topics of information theory include lossless/lossy data compression and channel coding.
1 Sliding Window
2 Input Block 1
Information theory stud
ies the transmission, proc
xtraction of information.
Entropy is usually express
6 Huffman Tree 1 Encoding
7 Huffman Tree 2 Encoding
I <0, 0>
n <0, 0>
f <0, 0>
o <0, 0>
. <10, 88>
E <0, 0>
n <1, 18>
o <2, 26>
p <0, 0>
. <3, 68>
3 4 5
Huffman Tree 1
Huffman Tree 2
Compressed Data
Output Block 1
Output Block 2
Output Block n

Figure 6.2: An illustration of the default compression work-flow of *Deflate* algorithm (specification 1.3).

man Coding [Huf52]). The basic algorithm works as follows. It takes a stream of input and moves the data through a *Sliding Window* buffer (32KB by default) as shown in Figure 6.2 (Label 1). It starts by checking if there is any string with length up to 258 bytes in the *Sliding Window* matching the string starting from the current position (the position in the *Sliding Window* as indicated by Label 2 in Figure 6.2) backwards up to a length of 258 bytes maximum. The longest match wins. The matched string will be replaced by a literal $l$ and a pair of values, as illustrated in Figure 6.2 (Label 3, 4, and 5). The literal (Figure 6.2 <Label 3>) is the immediate succeeding character of the matched string. The first value of the pair is the length of the matched string $m$ (Figure 6.2 <Label 4>). The second value is the distance $d$ (Figure 6.2 <Label 5>) from the current position to the matched string. Then, the *Deflater* advances $m$ characters in the input stream. If there is no match, the algorithm moves one character forward. This process iterates until the current input buffer is exhausted or the algorithm decides to start a new block of input when the current Huffman trees become inefficient.

The output from *LZ77* is a set of literals and pairs of values "$l$ $<m, d>$". The *Deflate* algorithm splits them into two columns. The literal and value $m$ are encoded by a Huffman tree (Figure 6.2 <Label 6>), the distance $d$ is encoded by a separate Huffman tree (Figure 6.2 <Label 7>). Both encoding results are merged and prepended with the two Huffman trees (both of the Huffman trees will be further encoded by the Huffman

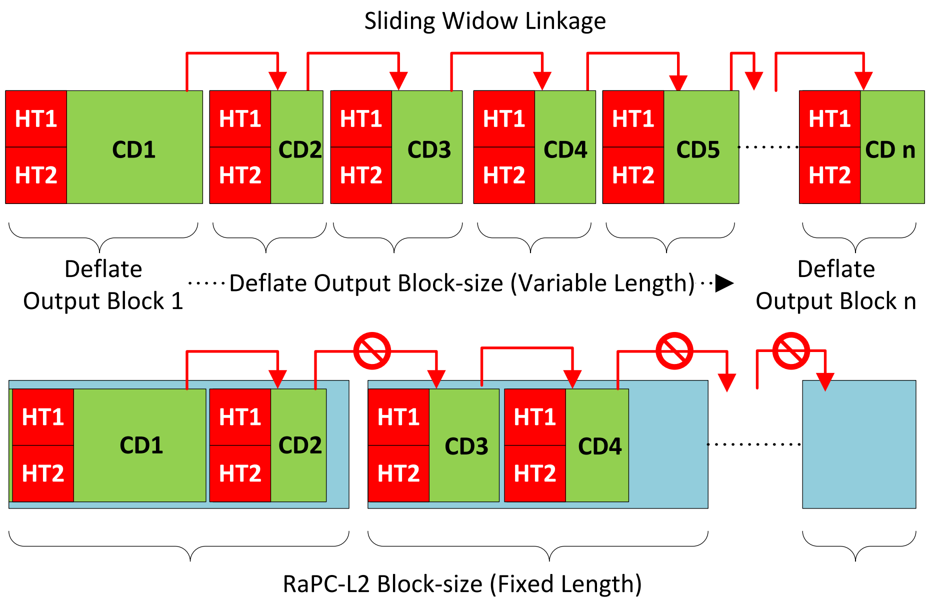


Figure 6.3: An illustration of RaPC Layer-2 block-packaging.

algorithm) for the final outputs as shown in Figure 6.2 (*Output Blocks*). Each output block corresponds to an input block. In principle, if each output block is self-contained and the boundaries of each block can be recorded, the default *Deflate* output data could be splittable to the HDFS. But, the *Deflate* algorithm tends to use the contents of the previous input buffer for the *Sliding Window* of its immediate succeeding input block in order to improve the compression ratio. These linkages between the input blocks create dependencies between the output blocks and cause the decompression process to be sequential as shown in Figure 6.3 (*upper*). If we simply remove these linkages, the overall compression ratio will degrade significantly.

In the RaCP-L2 scheme, we design a higher level buffer with a much larger and fixed size. We refer to this buffer as L2-Block and a *Deflate* output data block as DO-Block to avoid confusion. The L2-Block is used to accommodate a variable number of DO-Blocks. We force a break in the linkage between the last DO-Block in the current L2-Block and the first DO-Block in the succeeding L2-Block as illustrated in Figure 6.3 (*lower*). Thus, the compressed data in each L2-Block is completely self-contained (each L2-Block can be decompressed independently) which makes the RaPC-L2 splittable to the HDFS. The size of the L2-Block strikes a balance between the MapReduce analysis performance and the compression ratio. This will be further discussed in Section 6.3.

Moreover, recall that in the MapReduce framework, the *Record Reader* component supplies one logical record to a Mapper at a time. If we blindly compress data per block, it is very likely that the beginning and/or end of each *Deflate* input block (DI-Block) will contain incomplete record(s). An example is given in Figure 6.5 (*upper*). This makes the *Record Reader* complicated and inefficient at runtime in either of the two possible scenarios.

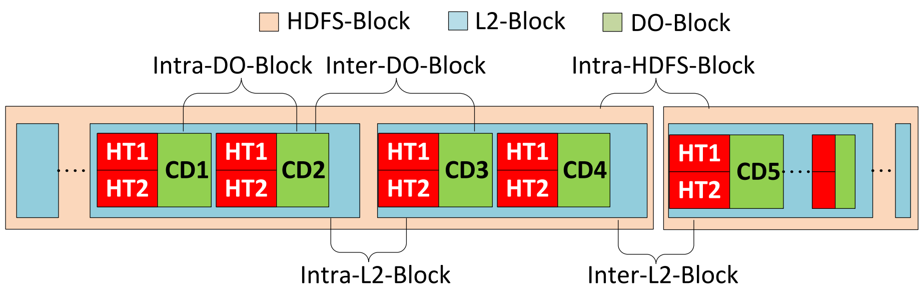


Figure 6.4: RaPC Layer-2 data organization using hierarchical structures.

- The first scenario (Intra-L2-Blocks) is about determining record completeness between two adjacent DO-Blocks in an L2-Block (Figure 6.4). In Hadoop, each HDFS-Block is processed by a dedicated Map process. The *Record Reader* that serves the Map process will firstly decompress one L2-Block (containing a number of Compressed Data <CD> blocks) at a time. Before fetching the decompressed data to the Map process, the *Record Reader* needs to determine whether the data contains partial records at both the beginning and end of the data. In fact, without decompressing the immediate succeeding L2-Block, we cannot be sure whether the last record is partial or complete. Therefore, we must remove and temporarily store the last record from the current decompressed data and wait until the next L2-Block is due for processing.
- The second scenario (Inter-L2-Blocks) is about determining record completeness between L2-Blocks (Figure 6.4). Recall that Hadoop HDFS organizes big files by splitting them into a series of fixed-size blocks (HDFS-Blocks in the context of this thesis). A series of HDFS-Blocks are distributed across the Hadoop cluster *Data Nodes*. There is no guarantee that logically adjacent HDFS-Blocks will be stored consecutively and/or on the same physical hard disk. Both the L2-Blocks (referring to Figure 6.4 *CD4* and *CD5*) belong to separate HDFS-Blocks and are possibly on different physical *Data Nodes*. The L2-Block (containing *CD5*) needs to be streamed to the current Map node (which is processing the L2-Block containing *CD4*) and decompressed. Then, record completeness must be checked as explained in Chapter 5. Both scenarios are time consuming and error-prone.

To remove this complication and improve the MapReduce work-flow efficiency, we ensure record completeness at the boundaries of each L2-Block during the data compression phase, hence making it record-aware. What constitutes a record is often dataset

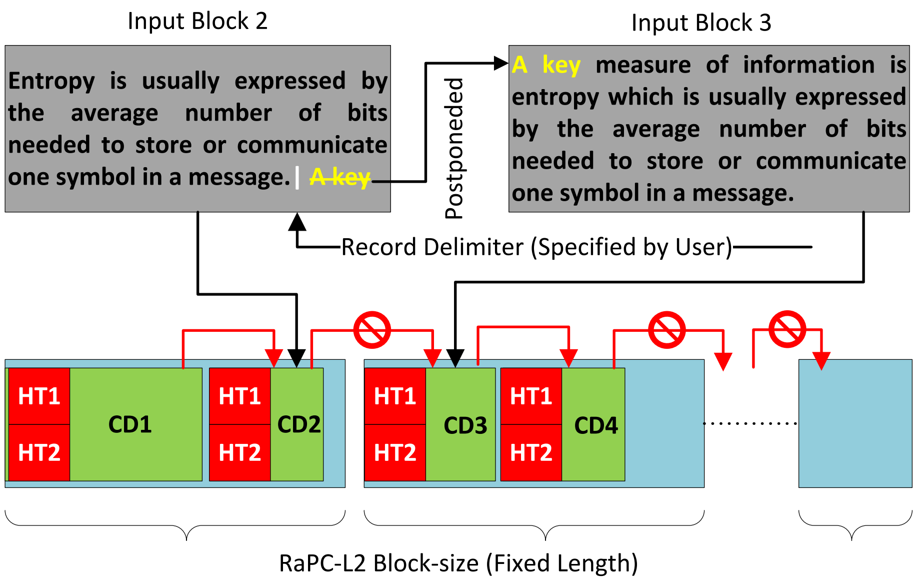

Input Block 2
Input Block 3
Entropy is usually expressed by the average number of bits needed to store or communicate one symbol in a message. | ~~A key~~
A key measure of information is entropy which is usually expressed by the average number of bits needed to store or communicate one symbol in a message.
Postponeded
Record Delimiter (Specified by User)
HT1
HT2
CD1
CD2
CD3
CD4
RaPC-L2 Block-size (Fixed Length)

Figure 6.5: RaPC Layer-2 record-aware compression scheme.

specific. To determine records, record delimiters must be given at the beginning of the data compression. At the last DI-Block (corresponding to the last DO-Block) of each L2-Block, we check for record completeness. Any partial record will be postponed to the next DI-Block which will eventually be packed into the next L2-Block. This process only occurs between DO-Blocks. An example is shown in Figure 6.5. Complex delimiters can be expressed in *Regular Expressions*. The compression scheme guarantees that each L2-Block contains a set of complete logical records.

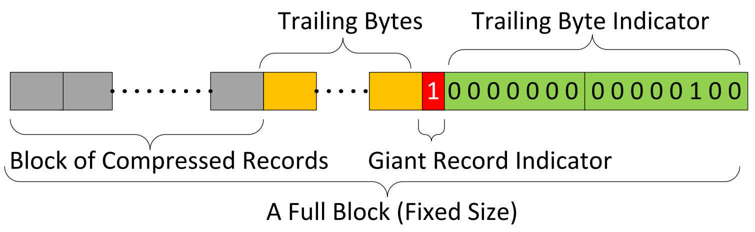


Figure 6.6: RaPC Layer-2 block-packaging scheme.

As we have noted, DO-Blocks have variable length and L2-Blocks have fixed size. When packaging a number of consecutive DO-blocks in a L2-Block, there is no guarantee that the L2-Block will be filled up exactly. We need to define a packaging format to tell RaPC how much of the payload is contained in the L2-Block. The detailed layout of the packaging format is illustrated in Figure 6.6. The gaps at the end of each L2-Block

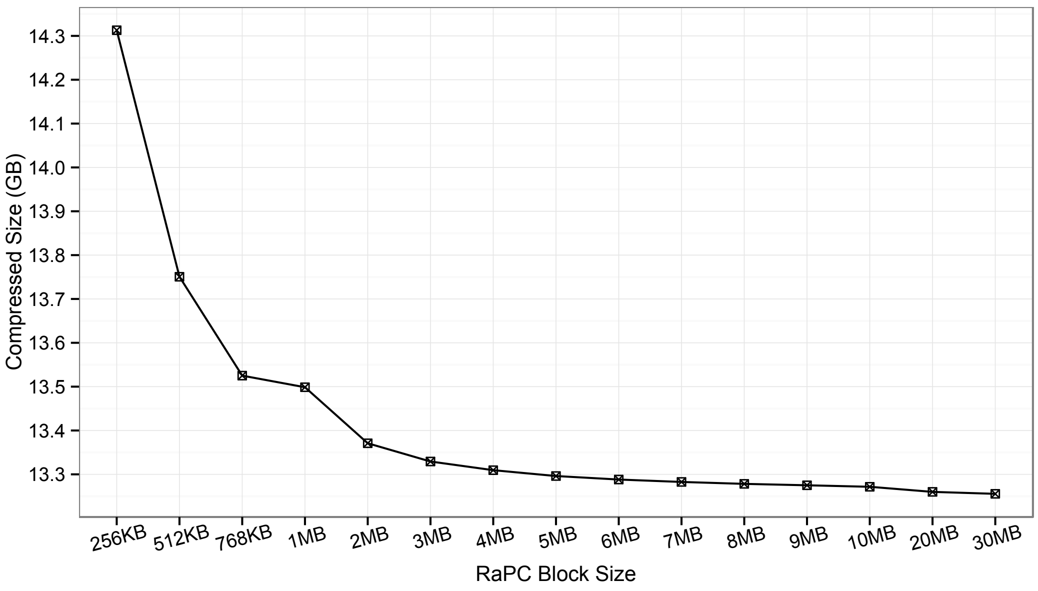


Figure 6.7: RaPC Layer-2 block size verses compression ratios

are filled by trailing bytes. The last two bytes of the L2-Block are reserved and used to indicate how many trailing bytes are used. In addition, the MSB of the second last byte is used to indicate whether there is a giant record that spans multiple L2-Blocks. This is designed for an occasion when a dataset contains very long records. Overall, the trailing byte indicator can only allow $2^{15} = 32$KB of trailing bytes to be inserted. In the case where there are only a small number of DO-Blocks left for packaging at the end of the compression, the number of trailing bytes needed may exceed this limit. For this reason, an exception is made so that the size of the last L2-Block is the total length of the last several DO-Blocks exactly. That is, no trailing bytes are inserted into the very last L2-Block.

Note that an increase in L2-Block size will slightly improve the compression ratio. Figure 6.7 shows the results from compressing dataset *DS-WikiEN* (the original dataset is 47GB, referring to Appendix A <Table A.2>) with various L2-Block sizes. In this experiment, we start with a L2-Block size of 256KB and gradually increase the block size to 30MB. In Figure 6.7, we observe a rapid drop in compressed data size (increased compression ratio) from 256KB to 2MB; and very slight drop in compressed data size when the L2-Block size is beyond 2MB. This is due to two reasons.

- Firstly, we break the Inter-DO-Block linkages to make the compressed data splittable. This implies that the current DI-Block cannot use the information from its immediate predecessor for the contents of the current *Sliding Window*. The disconnection leads to the compression for the current DI-Block to accumulate its own new context. At the beginning of this context accumulation phase, much less information can be compressed. This results in a lower compression ratio.

In addition, the *Sliding Window* is 32KB by default so that in the worst case scenario, if no information can be compressed during the context accumulation phase, there will be 32KB of incompressible data at the beginning of every L2-Block. When the L2-Block size is small, the 32KB incompressible data will occupy a relatively large proportion of the total ($\frac{32KB}{L2-Block\ size}$) and thus has a bigger influence on the compression ratio. This is the main cause of the rapid drop in the compressed data size (increase in compression ratio) as the L2-Block size increases.

- Secondly, during the L2-Block packaging processes, trailing bytes and indicator bytes must be appended to the last DO-Block to fulfil the requirements for a record-aware L2-Block. By increasing the L2-Block size, we have statistically reduced the number of trailing bytes and indicator bytes needed, thus improving compression ratio.

This case study gives us a general idea how the L2-Block size affects the compression ratio. The results in Figure 6.7 may vary slightly depending on the contents of a given dataset. Choosing a bigger L2-Block size can improve compression ratio, but it has side-effects on MapReduce program performance. This will be studied further in Section 6.3.

## 6.2 RaPC on Hadoop

In order to use RaPC compressed data on Hadoop with maximum transparency for MapReduce programs and developers, we provide a set of utility functions including: a customized *RaPC Record Reader* for decoding the RaPC-L2 compressed data; a *RaPC TextWritable* data type which implements equivalent operations found in the Hadoop *Text* data type for handling the RaPC-L1 compressed data; and a *SequenceFileAsBinaryOutputFormat Record Writer* for writing RaPC-L1 compressed data and general binary data to the HDFS.

The RaPC work-flow is as follows. The RaPC compressed files must be stored in the HDFS. If it is needed, the RaPC-L1 compression model file(s) must be loaded to the HDFS *Distributed Cache*. A MapReduce program must set the provided record reader (*RaPCInputFormat*) as the default input format class. This ensures that the RaPC Layer-2 compressed data can be correctly read and decoded. The input *value* type for Mappers must be set to *RaPCTextWritable* or the regular *Text* type. If the data is compressed by RaPC-L2(L1) (RaPC-L1 embedded in RaPC-L2), then the records received by the Mapper are in fact a block of binary data in the RaPC Layer-1 compressed format, then the *RaPCTextWritable* data type must be used. If the data is compressed by RaPC Layer-2 only, the decoded data is the original text, then the regular *Text* data type (in the context of the MapReduce framework) can be used.

For many data analysis jobs, parsing data records is necessary. For the original text, manipulating data is simple as the data contents are readable by developers. In RaPC, we provide a transformation function *T( )* that converts between original text and RaPC-L1 compressed data. For example, if we write a regular Java clause for searching a phrase "*value.contains("big data");*", then in RaPC, it is simply "*value.contains(RaPC.T("big data"));*". In this example, the variable "*value*" in the regular program can be a String data type. In RaPC, the variable "*value*" needs to be declared as a *RaPCTextWritable* data type. The "*contains( )*" function in RaPC is an implementation of the "*contains( )*" function found in the regular Java *String* class for the *RaPCTextWritable* data type. Indeed, the transformation requires the RaPC-L1 compression model file(s) to be loaded to the Map and/or Reduce during initialization. This may incur a small delay due to loading the model file(s) from HDFS *Distributed Cache* to the local computational nodes. For some type of jobs, for example, the *N-Grams* analysis, there is no need for model file(s), thus those MapReduce programs can take full advantage of using RaPC.

Another point worth mentioning is that the final MapReduce output data can be in the original text format or RaPC-L1 compressed format. In the former case, the RaPC-L1 decompression needs to be carried out at each Reducer before writing any results to HDFS. The second case needs more attention. By default, writing binary records to HDFS requires the *SequenceFileAsBinaryOutputFormat* provided by the MapReduce framework. This default *Record Writer* writes some auxiliary information about each binary record including record *key/value* length. It also periodically inserts synchronization points (*0xFF*) to the final output data. The record *key/value* length are directly converted from their numerical values to the byte sequences; this makes the final output data specific to Hadoop. Furthermore, the synchronization point is a character drawn from the extended ASCII codes which is very likely to clash with RaPC-L1 codes and confuse the RaPC-L1 decompression process. An additional requirement for writing the output data in RaPC-L1 compressed format to HDFS is therefore to use our customized *RaPCSequenceFileAsBinaryOutputFormat* which has those synchronization points removed and has well formatted *key/value* length.

Sample programs are given in Appendix C to demonstrate how the RaPC-L2 and RaPC-L2(L1)-enabled MapReduce programs differ from the same MapReduce programs with original text input as indicated by the highlighted code segments. In a RaPC-L2-enabled MapReduce program, configuring file input format is the only major difference; for an RaPC-L2(L1)-enabled program, we need to load the model file, configure the file input format, output format and corresponding data types.

## 6.3 Evaluation

We evaluate the effectiveness of RaPC using an on-site Hadoop cluster. The cluster's hardware specification and topology diagram are included in Appendix A (Table A.1 and Figure A.1). Specifically, the cluster is configured with Hadoop version 2.6.0 and Linux kernel 2.6.32. The HDFS is configured with single replication and 64MB block size. The MapReduce jobs used in the evaluation are summarized in Table 6.1. We use several real-world datasets listed in Appendix A (Table A.2) for the evaluation. The evaluations of the RaPC compressor and RaPC with MapReduce include measuring analysis performance, cluster storage requirements, memory constraints and compression performance. In the experiments, the number of Mappers and Reducers are optimized for each job. RaPC-L2 block-size was set to 1MB across all the experiments. We compare our approaches with the state-of-the-art Hadoop-LZO (also know as LZO-splittable, Twitter implementation), version 0.4.20.

### 6.3.1 Performance

Table 6.1 contains our main evaluation results. We use ten different MapReduce jobs to evaluate the effectiveness of the RaPC on Hadoop with a range of real-world datasets having different properties. We run each job three times with original, RaPC-L2 and RaPC-L2(L1) compressed data. We record the input data size, intermediate output data (Map output data) size, final output data (Reduce output data) size, memory allocation and analysis duration for each job with the three different input data types.

The *Site Rank* job is selected to demonstrate the full advantage of using RaPC with MapReduce. In this job, we calculate the rank for each website in the given dataset based on a ranking algorithm defined for an undirected graph. According to [Gro15], given an undirected graph *G(V, E)*, calculating the rank for vertex $v_i$ in *G* is equivalent to calculating the degree distribution of $v_i$ defined by $\frac{d(v_i)}{2|E|}$, where $d(v_i)$ is the degree of vertex $v_i$ in *G*. Thus, the site rank job is to obtain a vector as given by Formula 6.1, where the edges *E* are the connections between websites *V*.

$$SiteRank(G) = \frac{1}{2|E|} \begin{bmatrix} d(v_0) \\ d(v_1) \\ \vdots \\ d(v_n) \end{bmatrix} \tag{6.1}$$

In fact, the calculation part of the *Site Rank* job is relatively lightweight. The heavy part is parsing the dataset and finding out all of the connections between websites. The first iteration of the MapReduce job involves parsing the Uniform Resource Locator (URL) for each website, then finding and formatting edges *E*. A sample record from the

Table 6.1: A summary of MapReduce jobs used for evaluating the RaPC compression scheme.

| Job | Dataset | Input Type | Input | Intermediate | Output | Memory Allocation | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|---|
| Site Rank | DS-Memes | O : | 52.5 GB | 2.1 GB | 20.7 MB | 7.5 GB | 16m32s | | |
| | | H-LZO : | 21.0 GB | 2.0 GB | 20.7 MB | 4.1 GB | 8m56s | 46.0% | 60.0% |
| | | L2 : | 13.5 GB | 2.0 GB | 20.7 MB | 3.1 GB | 6m39s | 59.8% | 74.3% |
| | | L2(L1) : | 10.9 GB | 1.3 GB | 13.8 MB | 2.2 GB | 4m38s | 72.0% | 79.2% |
| 5-Gram | DS-WikiEN | O : | 47.0 GB | – | 150.9 GB | 12.7 GB | 25m19s | | |
| | | H-LZO : | 20.5 GB | – | 150.9 GB | 10.0 GB | 19m19s | 23.7% | 65.1% |
| | | L2 : | 13.4 GB | – | 150.9 GB | 8.8 GB | 16m52s | 33.4% | 71.5% |
| | | L2(L1) : | 10.9 GB | – | 87.6 GB | 7.1 GB | 16m18s | 35.6% | 76.8% |
| Word Count | DS-StackEX | O : | 40.6 GB | 30.6 GB | 9.0 GB | 24.1 GB | 74m39s | | |
| | | H-LZO : | 17.3 GB | 19.5 GB | 9.0 GB | 15.8 GB | 30m35s | 59.0% | 57.4% |
| | | L2 : | 11.5 GB | 19.1 GB | 9.0 GB | 14.7 GB | 28m33s | 61.8% | 71.7% |
| | | L2(L1) : | 9.6 GB | 16.4 GB | 6.3 GB | 13.7 GB | 26m22s | 64.7% | 76.4% |
| Publication Indexing | DS-PubMed | O : | 3.1 GB | 2.3 GB | 482.8 MB | 873.2 MB | 2m01s | | |
| | | H-LZO : | 2.1 GB | 2.2 GB | 482.8 MB | 777.4 MB | 1m47s | 11.6% | 32.3% |
| | | L2 : | 1.3 GB | 2.2 GB | 482.8 MB | 736.4 MB | 1m38s | 19.0% | 58.1% |
| | | L2(L1) : | 1.0 GB | 1.0 GB | 198.6 MB | 368.2 MB | 1m09s | 21.3% | 67.7% |

*continued . . .*

*. . . continued*

| Job | Dataset | Input Type | Input | Intermediate | Output | Memory Allocation | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|---|
| Music Rank | DS-Yahoo | O : | 10.2 GB | 235.6 MB | 3.2 MB | 3.4 GB | 6m48s | | |
| | | H-LZO : | 4.2 GB | 294.6 MB | 3.2 MB | 2.9 GB | 5m39s | 16.9% | 58.8% |
| | | L2 : | 2.4 GB | 166.1 MB | 3.2 MB | 2.3 GB | 4m35s | 32.6% | 76.5% |
| | | L2(L1) : | 2.2 GB | 156.6 MB | 2.8 MB | 2.4 GB | 4m37s | 32.1% | 78.4% |
| | | L2(L1-C) : | 2.2 GB | 156.6 MB | 3.2 MB | 2.5 GB | 4m45s | 30.1% | 78.4% |
| Customer Satisfaction | DS-Amazon | O : | 33.4 GB | 223.1 MB | 496.2 MB | 4.8 GB | 10m16s | | |
| | | H-LZO : | 17.1 GB | 194.2 MB | 496.2 MB | 3.2 GB | 6m50s | 33.4% | 48.8% |
| | | L2 : | 11.1 GB | 183.3 MB | 496.2 MB | 2.0 GB | 4m24s | 57.2% | 66.8% |
| | | L2(L1) : | 8.3 GB | 154.8 MB | 350.7 MB | 3.5 GB | 6m48s | 33.8% | 75.1% |
| | | L2(L1-C) : | 8.3 GB | 154.8 MB | 496.2 MB | 3.5 GB | 6m54s | 32.8% | 75.1% |
| Data Preprocessing | DS-Memes | O : | 52.5 GB | 15.1 GB | 29.5 GB | 7.9 GB | 17m56s | | |
| | | H-LZO : | 21.0 GB | 15.0 GB | 29.5 GB | 4.8 GB | 11m33s | 35.6% | 60.0% |
| | | L2 : | 13.5 GB | 15.0 GB | 29.5 GB | 3.7 GB | 8m46s | 51.1% | 74.3% |
| | | L2(L1) : | 10.9 GB | 11.0 GB | 16.3 GB | 2.3 GB | 5m57s | 66.8% | 79.2% |

*continued . . .*

*. . . continued*

| Job | Dataset | Input Type | Input | Intermediate | Output | Memory Allocation | Duration | Performance Gain | Size Reduction (Input) |
|---|---|---|---|---|---|---|---|---|---|
| Format Conversion | DS-Twitter | O : | 17.3 GB | – | 30.4 GB | 3.2 GB | 6m34s | | |
| | | H-LZO : | 10.0 GB | – | 30.4 GB | 2.5 GB | 5m01s | 23.6% | 42.2% |
| | | L2 : | 6.5 GB | – | 30.4 GB | 2.0 GB | 3m58s | 39.6% | 62.4% |
| | | L2(L1) : | 6.1 GB | – | 28.1 GB | 1.6 GB | 3m18s | 49.7% | 64.7% |
| Event Identification | DS-Google | O : | 158.9 GB | 282.1 MB | 32.7 KB | 18.1 GB | 39m52s | | |
| | | H-LZO : | 60.9 GB | 108.7 MB | 32.7 KB | 9.5 GB | 19m55s | 50.0% | 61.7% |
| | | L2 : | 36.0 GB | 64.8 MB | 32.7 KB | 6.4 GB | 13m12s | 66.9% | 77.3% |
| | | L2(L1) : | 31.1 GB | 56.6 MB | 20.9 KB | 8.4 GB | 16m41s | 58.2% | 80.4% |
| | | L2(L1-C) : | 31.1 GB | 56.6 MB | 32.7 KB | 8.4 GB | 16m49s | 57.8% | 80.4% |
| Server Log Analysis | DS-Google | O : | 158.9 GB | 1.0 GB | 371.1 KB | 26.3 GB | 54m57s | | |
| | | H-LZO : | 60.9 GB | 414.1 MB | 371.1 KB | 14.6 GB | 29m08s | 47.0% | 61.7% |
| | | L2 : | 36.0 GB | 246.8 MB | 371.1 KB | 11.4 GB | 22m26s | 59.2% | 77.3% |
| | | L2(L1) : | 31.1 GB | 196.2 MB | 273.5 KB | 12.1 GB | 23m16s | 57.7% | 80.4% |
| | | L2(L1-C) : | 31.1 GB | 196.3 MB | 371.1 KB | 12.5 GB | 24m04s | 56.2% | 80.4% |

O: Original dataset; H-LZO: Hadoop-LZO; L1: RaPC Layer-1 compression; L2: RaPC Layer-2 compression. The final output is in original format; L2(L1): RaPC Layer-1 embedded in RaPC Layer-2 compression. The final output is in RaPC Layer-1 compressed format; L2(L1-C): RaPC Layer-1 embedded in RaPC Layer-2 compression. The final output is in original format. *1GB = 1,073,741,824 Bytes* dataset is shown in List 6.1. For this particular dataset, each record is separated by an empty line. Lines starting with "*P*" indicate the source Web link. The lines starting with "*L*" indicate the Web links that are referenced by the source Web link. Lines starting with "*T*" and "*Q*" need to be ignored in this job. The head and tail of each Web link (highlighted in shaded grey in List 6.1) need to be trimmed off. The *key/value* pairs in the Map output data for this sample record is given in Table 6.2. Then, we will receive $d(v_i)$ at the Reduce phase. In the second iteration, the duplicate links generated from the previous iteration need to be removed for the purpose of calculating $|E|$. Then finally the *SiteRank(G)* can be obtained.

Listing 6.1: A sample record from the *Memetracker memes* dataset [LBK09]

```
P    http://dotshout.com/news/view.php?post_id=216331
T    2008-08-01 00:00:29
Q    the impact of the economic downturn is really twofold
Q    we encourage everybody to get onto the same page
L    http://gizmodo.com/tag/early-termination-fees
L    http://gizmodo.com/tag/etf-fees
L    http://mercurynews.com/ci_10039461?source=most_viewed
L    http://tech.yahoo.com/blogs/null/99655
```

Table 6.2: Examples of the intermediate outputs (results for the Map processes) from parsing the *Memetracker memes* dataset.

| Record Index | Key | Value |
|---|---:|---|
| 1 | dotshout.com | gizmodo.com |
| 2 | dotshout.com | gizmodo.com |
| 3 | dotshout.com | mercurynews.com |
| 4 | dotshout.com | tech.yahoo.com |
| 5 | gizmodo.com | dotshout.com |
| 6 | mercurynews.com | dotshout.com |
| 7 | tech.yahoo.com | dotshout.com |

Using the RaPC-L2 compressed data for the *Site Rank* job, we accelerated the calculation speed by 59.8% compared to the same job with the original text data. The performance gains come from two sources. Firstly, the RaPC-L2 compressed data is approximately four times smaller than the original data (RaPC-L2 reduces the original data size by 74.3%). Loading the RaPC-L2 compressed data from hard disks to memory requires much less time. Secondly, due to the MapReduce *Local Combiner* effects, the intermediate data is smaller. This reduces the time required to transmit data from Mappers to the Reducers over the network.

In addition to using RaPC-L2 compressed data, using RaPC-L2(L1) compressed data can further improve the analysis performance. When parsing the URLs, we need to split each URL using the forward slash character "/". The splitting result gives an array of sub-strings drawn from the Web link. We are interested in the website link part only. The access protocol and the page identifier parts must be omitted. The string splitting process is about searching for the delimiter "/" and taking the sub-string from the link iteratively. By default, the Java implementation of the string *split* function invokes the *indexOf( )* function to locate patterns which takes $O(m(n-m))$ time, where $m$ and $n$ are the pattern length and source string length, respectively. In this case, $m = 1$, therefore each search on pattern "/" takes $O(n-1)$ time. Recall that the RaPC-L1 compression does not compress functional contents. We can search for the forward slash character from the RaPC-L1 compressed data directly without using the model files. The compression ratio for this particular dataset can be found in Figure 6.13. It is approximately 49%. This implies searching the same pattern from the new source $O(n'-1)$ is approximately twice as fast as $O(n-1)$, where $n' = 0.49n$.

In the second job, we use a *N-gram* task for evaluating the RaPC under intensive disk I/O operations. *N-gram* analysis is a common technique for speech recognition. We use 5-Gram to produce sufficient in-memory data to increase the frequency of memory to disk I/O operations. The job consists of Map tasks only. There are no *Partitioning*, *Shuffling* and *Reducing* phases, thus the network I/O is kept to a minimum. Comparing the results for the original and RaPC-L2 compressed data, RaPC-L2 improves the processing speed by 33.4%. Because both analyses generate the same output data size, therefore the main performance gain is from distributing MapReduce data splits to Mappers. Using RaPC-L2(L1) compressed data can further improve the analysis speed by $\sim$2.2% because of the smaller final output data in RaPC-L1 compressed format.

Besides the efficiency, there is a useful side-effect when using RaPC-L2(L1) compressed data. Recall that the *5-Gram* job reads, manipulates and writes data in the RaPC-L1 compressed format. During the lifetime of the job, the RaPC-L1 model files are not required. As the RaPC-L1 compression *encrypts* (encodes) the informational contents, using the RaPC-L2(L1) compressed data can provide a certain level of protection on data privacy in a shared cluster or in a public cloud environment.

The *Word Count* job is used as a standard benchmark for results comparison. Also, the *Word Count* job generates comparatively larger intermediate and final output data. Using RaPC-L2 or RaPC-L2(L1) compressed data, we can reduce the size of the intermediate output data by 33.5% and 43.5%, respectively. Thus, the transport cost of distributing the Map output data to corresponding Reducers over the network (MapReduce *Shuffling* phase) can be reduced significantly. It also reduces the cost of materializing in-memory data to local persistent storage (local hard disks). The reduction of the size of the intermediate output data is due to the MapReduce *Local Combiner* effects.

Recall the work-flows in the Map phase and note that the following parameters and figures are the default values in Hadoop version 2.5.0 or above. When processing a data split, the Map output data is temporarily stored in an in-memory buffer (100MB). When the buffer fills to a certain threshold (80%), a background process starts materializing the in-memory data to the local persistent storage. During this *spilling* phase, the buffered data is firstly partitioned according to the number of Reducers configured for this particular job. Within each partition, records are sorted by *keys*. The sorting results are then fetched to the *Local Combiner* process to combine the records with duplicate *keys*. The output data from the *Local Combiner* is called a "*spill*" and it is then written to local hard disks. Generally, the data processed by a *Local Combiner* is much smaller depending on the number of records with duplicate keys found. Thus, we can reduce the time used to write *spill* data to disks.

For each individual Mapper, if the generated intermediate data is larger than the in-memory buffer size, each Mapper will produce a series of *spills*. Upon the completion of processing a data split, all partitions from the group of *spills* that belong to the same Mapper will be merged based on their partition number and then sorted by record *keys*. The sorting results are again fetched to the *Local Combiner* to further examine records with duplicate *keys*. This is the place where the RaPC can reduce the intermediate data size. The reasons are as follows. In any given cluster environment, the HDFS-Block size is fixed. Assume processing each data split will generate $m$ records on average. With the original text data, the *Local Combiner* process is about finding records with duplicate *keys* in $m$. With RaPC compressed data, each Mapper will receive a block of data with the same size, but in compressed format. Given the compression ratio $\varphi$, defined as $\varphi = \frac{S_o - S_c}{S_o}$ (Section 1.5), each Mapper will actually receive $\frac{S_c}{1-\varphi}$ sized data. If $m$ increases proportionately with the data split size $m \propto S_o$, then the *Local Combiner* will search records from the RaPC compressed data with duplicate keys in $\frac{S_c}{(1-\varphi)S_o}m$. For example, in this particular job, using the original text data, each Mapper will receive 64MB (equal to the HDFS-Block size) data split on average and produce $m$ records. For the RaPC-L2 compressed data with a compression ratio of 71.7%, each Mapper will receive $\frac{64MB}{(1-0.717)} \approx$ 226MB data (after RaPC-L2 decompression) and produce $\sim$3.5$m$ number of records. When $m$ increases, we will have a statistically greater chance of finding more records with duplicate keys and/or more duplicate keys for a record, thus further reducing the size of the intermediate outputs. In general the bigger the size of L2-Block, the more the effects of *Local Combiner* with RaPC are further enhanced.

In the *Publication Indexing* job, we calculate the descriptive statistics for each publication based on the importance of the author(s) which is further defined by the number of publications from the author. Thereafter, the index of the publication can be sorted by designated statistics. We use the *PubMed* database [LAB$^+$09] for this job. The dataset contains $\sim$22 million publications and $\sim$11 million authors. Referring to Table 6.1, RaPC-L2 compresses the data by 58.1% which is the lowest ratio among the ten jobs.

Considering the similar intermediate and final output data size, this low compression ratio leads directly to the low performance gains of 19%. Further improvement of 2.3% is given by using the RaPC-L2(L1) compressed data which is the combined contribution from the smaller intermediate/final data size and faster string manipulation speeds.

Table 6.3: Comparison of the number of Map input records with respect to the original, RaPC-L2 compressed, and RaPC-L2(L1) compressed data.

| | Number of Map Input Records | | |
|---|---|---|---|
| Job | Original | RaPC-L2 | RaPC-L2(L1) |
| Site Rank | 919061195 | 13810 | 11160 |
| 5-Gram | 805750261 | 13823 | 11027 |
| Word Count | 6126845962 | 11745 | 9881 |
| Publication Indexing | 21788173 | 1370 | 1020 |
| Music Rank | 699640226 | 2435 | 2268 |
| Customer Satisfaction | 381554470 | 11397 | 8490 |
| Data Preprocessing | 919061195 | 13810 | 11160 |
| Format Conversion | 468854886 | 6622 | 6232 |
| Event Identification | 1232799308 | 36915 | 31840 |
| Server Log Analysis | 1232799308 | 36915 | 31840 |

In the *Music Rank* job, we use the *Yahoo! Music Rating* dataset [Yah06] which contains a large number of very small records (approximately 16 characters per record on average). The dataset contains $\sim$700 million ratings on $\sim$136 thousand songs provided by *Yahoo! Music* services. The task is to calculate the mean scores and standard deviation for each song. For this task, the MapReduce *Record Readers* are heavily loaded supplying records to the corresponding Mappers. The default *Record Reader* provided by the MapReduce framework treats a line as a record. The records processed by a Mapper will have on average 16 characters. This makes the default *Record Reader* inefficient and wastes a lot of cluster resources such as Java Heap Space (memory) assigned to the Mapper. With compressed contents, our *RaPCRecordReader* is in fact supplying a set of records to a Mapper at a time. Table 6.3 shows the number of records received by Mappers for each job with different input data types. It works by reading a fixed-size block of data (L2-Block, the size of L2-Block is adjustable). The block of data will be decompressed in-memory and then forwarded to a Mapper. Because the L2-Block size is fixed and more importantly, each L2-Block contains a set of complete records, there is no need to track record length and worry about partial records at the boundaries of L2-Blocks and HDFS-Blocks. This makes the *RaPCRecordReader* more efficient and lightweight than the default MapReduce *Record Reader*. Additional to the RaPC-L2(L1) compressed data, in order to calculate *music scores*, we must convert the text contents to real numerical values. This conversion requires loading the RaPC-L1 compression model files to each Mapper. The loading process and the text to numerical value conversion take extra time and occupy more memory. This leads to the job with the RaPC-L2(L1) compressed data being slightly slower than the job with RaPC-L2 compressed data.

The *Customer Satisfaction* job is a special case. When processing RaPC-L2 compressed data, the job results in a relatively high performance gain of 57.2%. Two sources contribute most to this result. The first source is the lower data transmission cost due to the smaller data size. The second source is the *Group Record Effect*. Records in this particular dataset are split by empty line(s). Each record consists of multiple fields delimited by a new line character. Records contain a variable number of fields. By default, the MapReduce framework treats the new line character as the delimiter. The default *Record Reader* supplies a line of text to a Mapper at a time. This requires Mappers to wait for the *Record Reader* until a full logical record is received. The waiting time is the main cause of the delay. With RaPC compressed data, a group of complete records are given to one Mapper at a time, therefore it saves on waiting time. Indeed, writing a customized *Record Reader* using a traditional approach for this particular dataset can improve the analysis performance on the original dataset. But, it will incur extra programming and will still suffer from the big data size.

The *Data Preprocessing* job is used to evaluate the effectiveness of RaPC-L2(L1) when dealing with highly skewed giant records. In this job, we create records that have a source website link as the *key* and all web pages that refer to the source website as the *value*. The preprocessed data thereafter can be used for other calculations, for example, web site popularity and centrality analysis. We use the *Memetracker memes* dataset [LBK09] in this experiment. The dataset contains ∼418 million web page URLs. Some extremely popular websites are referenced by many web pages. Formatting and appending the referring web page links to a source website can result in a giant record. If the size of the giant record is larger than the size of the Java Heap Space that is currently available to the Mapper, it will cause very frequent in-memory data to disk swapping processes. This event was captured and shown in the shaded area in Figure 6.8 during the task executions. The disk I/O bursts occurred in both the tasks with original text and RaPC-L2 compressed text. We have observed that the bursty levels are similar in the two tasks shown in Figure 6.8 (*upper*) and (*middle*). This is because both tasks process the text in its original format. The Map output data must be identical for both jobs regarding the data contents and length. This irregular event also causes extra delay to the job completion time. Note that these big peaks occurred during the Reducing phase while writing the final output data to the HDFS (Figure 6.8).

In contrast, when processing RaPC-L2(L1) compressed data, it is hard to identify any obvious disk I/O bursts in Figure 6.8 (*lower*). Referring to Figure 6.13, the RaPC-L1 compression ratio for this particular dataset is ∼49%. The length of the records in

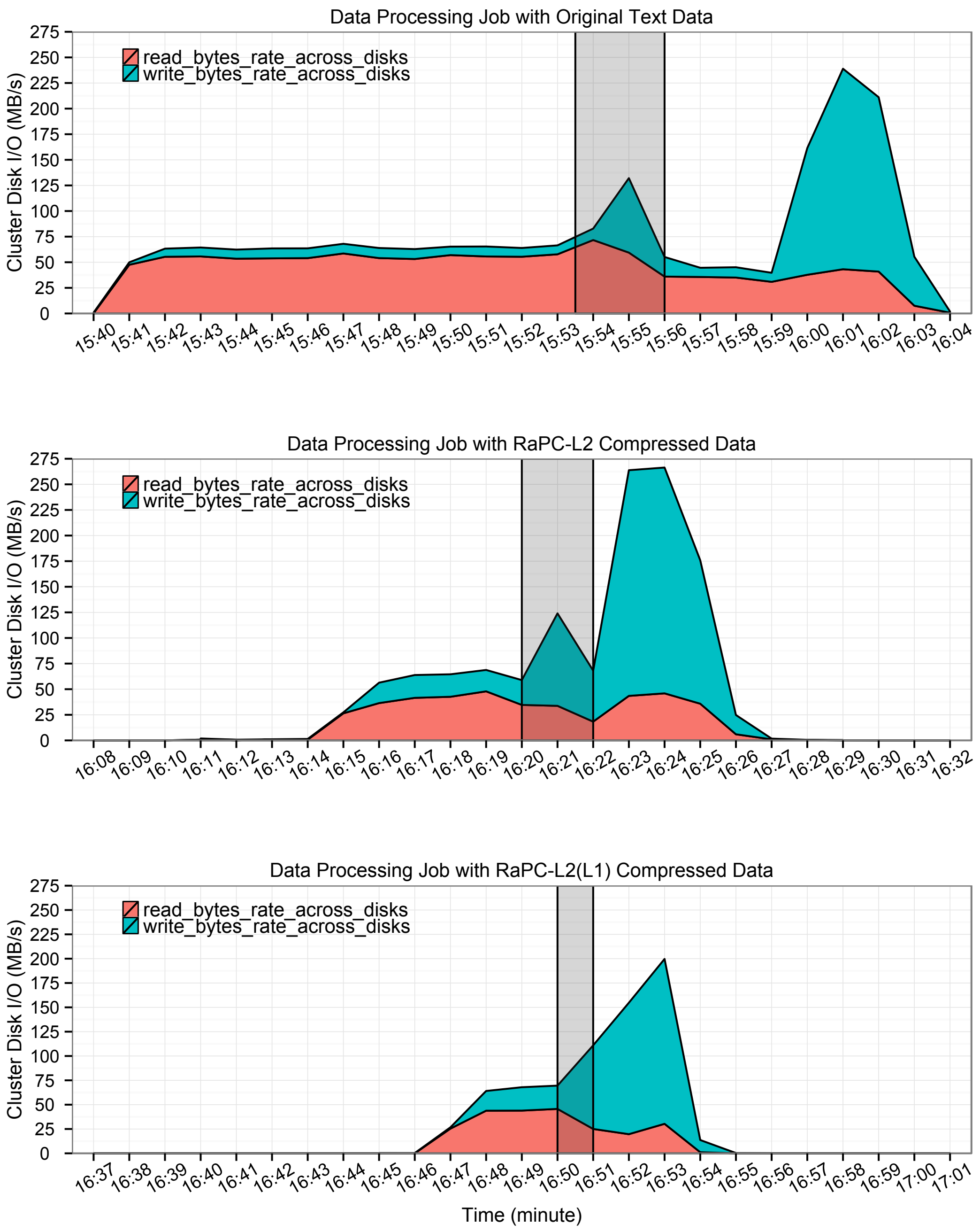


Figure 6.8: The *Data Preprocessing* job generates highly skewed records causing a disk I/O burst due to swapping.

RaPC-L1 compressed format is approximately half the length of those produced by the same tasks with either original or RaPC-L2 compressed text.

Additionally, in the previous version of Hadoop, when the record size is larger than the size of the Java Heap Space available to the Mapper, the MapReduce framework will throw *Java Heap Space Errors*. To solve this, the corresponding parameter "*mapred.reduce.child.java.opts*" needs to be re-adjusted to a bigger value and then the entire cluster needs to be restarted to pick up the changes.

The *Format Conversion* job is similar to the previous task. In this job, we convert the Twitter *tweets* from Tab Separated Values (TSV) format to the JavaScript Object Notation (JSON) format. The basic process is to split each *tweet* by the functional character "Tab". The separation produces a group of strings which will be treated as attribute values in a JSON object. Values will be prepended with corresponding attribute names and enclosed with a pair of curly brackets to form a legitimate JSON object. Again, this task does not require the RaPC-L1 compression model file(s). The main purpose of this job is to zoom in on the differences between the MapReduce tasks with RaPC-L2 and RaPC-L2(L1) compressed data. Referring to Table 6.1, using RaPC-L2(L1) compressed data further improves the processing speed by 11.1% comparing to RaPC-L2. Notice that both jobs have a similar input and output data size, thus the time saved on reading and writing data contributes a small portion to the overall improvement. The majority of the contributions are from manipulating RaPC-L1 compressed data. Based on the previous analysis, the improvement can be roughly estimated using the RaPC-L1 compression ratio. For this particular job, the job with RaPC-L2 and RaPC-L2(L1) data took 288 and 235 seconds, respectively. If we assume that the major part of the data processing is the pattern searching, using the RaPC-L2(L1) compressed data can approximately reduce the searching time by $\varphi_{L1}$. Referring to Figure 6.13, the $\varphi_{L1}$ for dataset *DS-Twitter* is ~19%. Thus, the performance gain from using the RaPC-L2(L1) data can be estimated by $288 - 288 * \varphi_{L1} = 233.28$ which is close to the observed 235 size.

The *Server Log Analysis* and the *Event Identification* jobs are used to evaluate RaPC with comparatively large jobs. The main task of the *Server Log Analysis* is to determine the over/under utilized servers from *Google Cluster Log* files [CRH11]. The dataset contains ~1.2 billion records in Comma Separated Value (CSV) format. For this particular dataset, RaPC-L2 and RaPC-L2(L1) can reduce the original data size by 77.3% and 80.4%, respectively. The size of the intermediate output data is significantly smaller because of the MapReduce *Local Combiner* effects discussed above and the highly repetitive nature of the data. Due to the significant data size reduction (both input and intermediate output), we have achieved 59.2% and 57.7% performance gains from the job with RaPC-L2 and RaPC-L2(L1) compressed data, respectively. During the analysis, numerical values in the RaPC-L1 format need to be decoded to standard strings and then converted into real numerical values. This requires the RaPC-L1 compression model file to be available to each Mapper, which makes the RaPC-L2(L1) job slower than the job with the RaPC-L2 compressed data. The level of the performance degradation is influenced by the number of Mappers and how much RaPC-L1 compressed data needs to be decoded during the analysis. If we increase the HDFS-Block size (the same as reducing the number of Mappers), we can improve analysis speed and memory consumption, accordingly. Note that the RaPC-L1 compression model file(s) needs to be loaded to each Mapper separately. The smaller number of Mappers results in less memory (aggregated) being used to hold the RaPC-L1 model file(s). Rationally, if a considerable portion of the RaPC-L1 data needs to be decoded during the job execution, then just using the RaPC-L2 compression for the data is a more appropriate approach. The same reasoning also applies to the *Event Identification* job.

In the *Event Identification* job, we aim to identify the abnormal events that occur on each cluster node and produce a list of the unhealthy nodes with the event IDs and time stamps. In addition to the three jobs with original, RaPC-L2 and RaPC-L2(L1) compressed data, we also run an additional job RaPC-L2(L1-C) with the final output data in the original format. This requires loading the RaPC-L1 compression model file(s) to Reducers which will take extra time and delay the completion of the entire job.

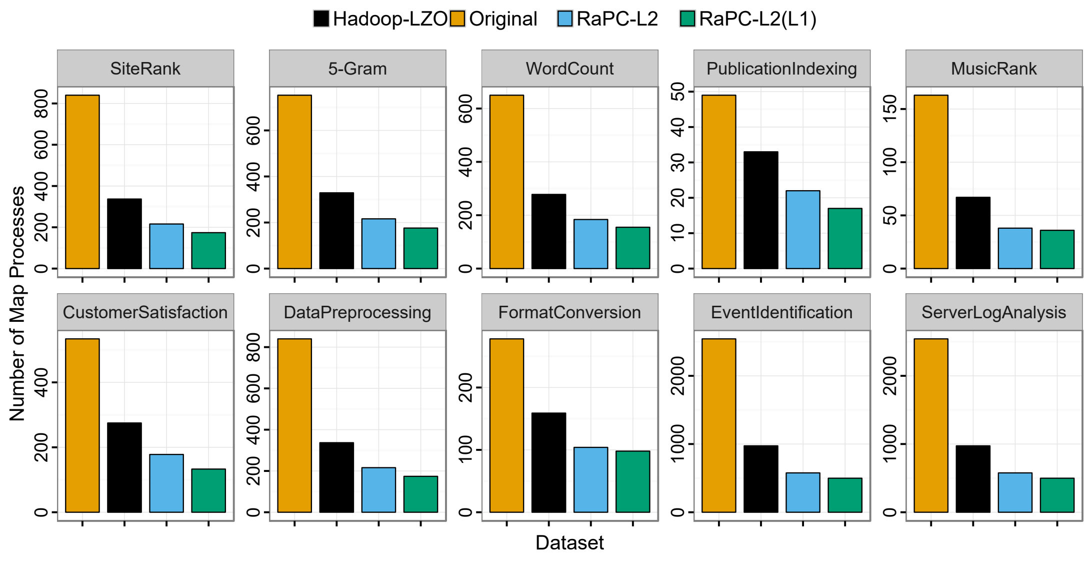


Figure 6.9: Number of Mapper processes configured for each job with original, RaPC-L2 compressed, and RaPC-L2(L1) compressed data, respectively.

In general, using the RaPC-L2 or RaPC-L2(L1) compressed data requires much less memory. The reasons are as follows. Firstly, Mappers are independent processes and require dedicated Java Virtual Machines (JVMs). Each has a default and dedicated Java Heap Space assigned to it. The more Mappers the more memory is required. Because the input data size is the dominant factor affecting the number of Mappers, the compressed data size is significantly smaller resulting in a smaller number of Mappers and consequently less aggregate memory consumption. Figure 6.9 shows the default number of Mappers configured for each job with the original, RaPC-L2 and RaPC-L2(L1) compressed data. Moreover, supplying a block of data to a Mapper at a time can make more efficient use of memory .

The following applies to the RaPC-L2(L1) compressed data. At a lower level, text data

Table 6.4: Performance of the Hadoop-LZO compression.

| Dataset | Original Data Size | LZO Compressed Data Size | Compression Time | Indexing Time | Index Size |
|---|---|---|---|---|---|
| DS-Amazon | 33.4 GB | 17.1 GB | 6m44s | 6m46s | 1.0 MB |
| DS-Google | 158.9 GB | 60.9 GB | 30m17s | 10m04s | 5.1 MB |
| DS-Memes | 52.5 GB | 21.0 GB | 10m17s | 3m53s | 1.6 MB |
| DS-PubMed | 3.1 GB | 2.1 GB | 0m38s | 0m54s | 98.0 KB |
| DS-StackEX | 40.6 GB | 17.3 GB | 8m19s | 2m53s | 1.3 MB |
| DS-Twitter | 17.3 GB | 10.0 GB | 3m34s | 4m10s | 555.0 KB |
| DS-WikiEN | 47.0 GB | 20.5 GB | 8m38s | 4m41s | 1.5 MB |
| DS-Yahoo | 10.2 GB | 4.2 GB | 1m50s | 0m45s | 325.0 KB |

is often held by a String (in Java) or Text (in MapReduce) object. Each String or Text object will have an internal buffer to hold the actual data and the length of the buffer is determined by the length of the record to be held, with some redundancy in this case. During the analysis, since records are often split, re-formed or concatenated, a large number of such temporary objects will be created and disposed of. This has negligible effects on the overall aggregate memory consumption. With RaPC-L1 compressed data, the average length of temporary objects can be shortened. Reducing memory consumption can be particularly useful for a multi-tenant cluster in which the saved memory can be assigned to more users.

Furthermore, we compare the RaPC scheme with the state-of-the-art Hadoop-LZO. Using Hadoop-LZO compressed data in Hadoop is similar to our RaPC scheme. Hadoop-LZO compresses data using the standard LZO compressor. The compressed data is then uploaded to HDFS. A special indexer program is used to index and record the splittable boundaries of the compressed data. The output of the indexer is a set of indexing files which are associated with each individual file in the given dataset. In MapReduce, each Mapper uses the indexing files to calculate splittable boundaries and take the data block for processing. Table 6.4 shows the performance of the Hadoop-LZO, in terms of compression ratio, compression speed and the size of the indexing file(s). As the number of files increases, maintaining and processing the indexing files can be complicated. The main comparison results are shown in Table 6.1. In general, using RaPC compressed data can improve MapReduce performance by 13.5% (RaPC-L2) and 15.1% (RaPC-L2(L1)), further reduce data size by 16.2% (RaPC-L2) and 21.0% (RaPC-L2(L1)), on average, compared to that using Hadoop-LZO compressed data.

In brief, we have demonstrated the flexibility and effectiveness of using RaPC with MapReduce using a variety of standard MapReduce jobs. Using RaPC-L2 does not exhibit any shortcomings and RaPC-L2(L1) can further improve efficiency of analysis

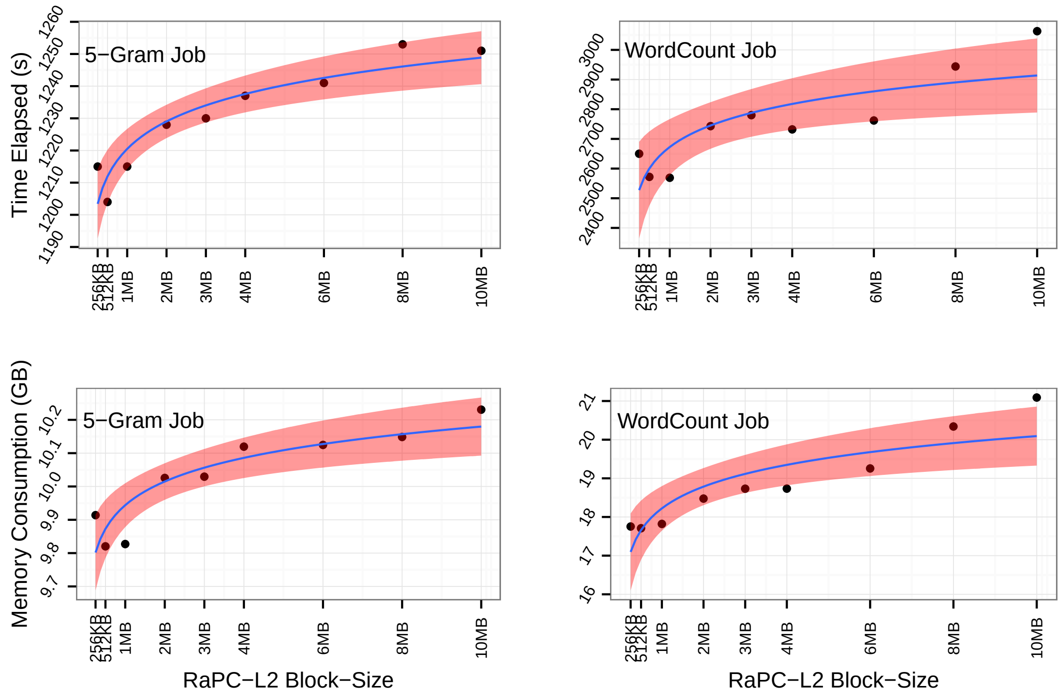


Figure 6.10: RaPC-L2 block-size verses analysis performance and memory consumption.

speed, cluster memory usage and storage usage. It can be most beneficial to clusters that are I/O bound and for which storage space is a concern. For some jobs that need many conversions, for example, an analysis on financial reports, using the *T( )* function frequently can affect the overall performance. In those cases, using RaPC-L2 compression alone may be more appropriate.

### 6.3.2 RaPC-L2 Block Size Effect

As we have seen in the previous section, using RaPC-L2 or RaPC-L2(L1) compressed data can improve MapReduce program performance and decrease memory consumption. By default, we use a 1MB L2-Block for all experiments given in Table 6.1. In this section, we aim to identify the relationship between the L2-Block size, analysis performance and memory consumption. Note that the L2-Block size is the size of a block of data supplied to a Mapper at a time. Increasing the L2-Block size is equivalent to increasing the workload for each individual Mapper at a time. In the experiments, we compress dataset *DS-WikiEN* with different L2-Block sizes from 256KB to 10MB (when the L2-Block size is greater than 10MB, the MapReduce framework starts reporting *Java Heap Space Errors* as limited by the current cluster configuration). We use *5-Gram* and *Word Count* jobs as two test cases. These jobs are then performed giving each compressed data different L2-Block sizes. We keep the same number of Reducers and use an

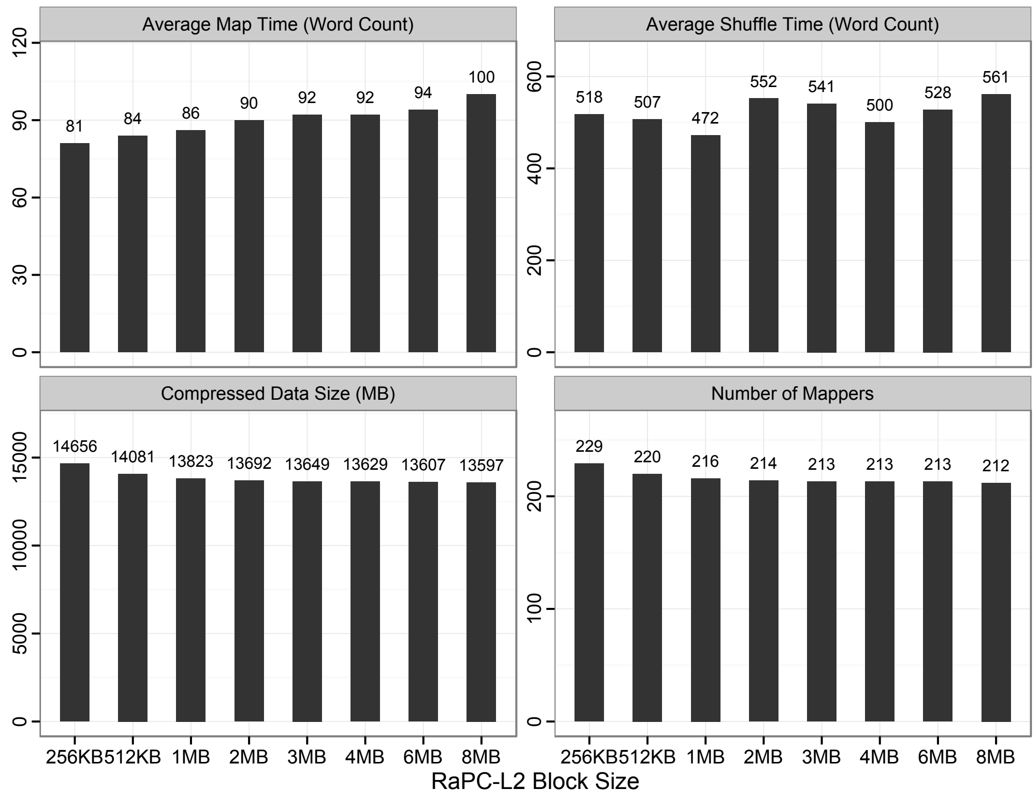


Figure 6.11: RaPC-L2 block-size verses analysis performance and memory consumption for the *Word Count* job.

optimal number of Mappers for each run. Figure 6.10 shows very interesting results. Increasing the L2-Block size slightly decreases the MapReduce program performance and increases memory consumption.

In particular, we observe a performance and memory usage degradation when L2-Block size increases from 256KB to 1MB for both the *5-Gram* and the *Word Count* jobs. To find the root cause, referring to Figure 6.7 again, there is a sharp drop in compressed data size when the L2-Block size increases from 256KB to 1MB. This reduction in compressed data size leads to a change in the number of Mappers per MapReduce program. When the L2-Block size is above 1MB, the compression ratio tends to be constant. The exact values of the compressed data size and the corresponding number of Mappers are shown in Figure 6.11 for the *Word Count* job. We have already identified that an increase in the number of Mappers implies more aggregate memory needed and more time required for loading data from disks to memory. The change in the number of Mappers is one of the main reasons for the sudden increase in performance and the decrease in memory consumption from 256KB to 512K/1M.

Starting from an L2-Block size of 1MB, the changes to the number of Mappers becomes insignificant as shown in Figure 6.11 (*bottom-right*). The increase in memory consumption is now due to the internal buffer size required to hold the decompressed data at the Mappers and the *Record Readers*. Given that the RaPC-L2 compression ratios for dataset *DS-WikiEN* for L2-Block sizes of {1MB, 2MB, 3MB, 4MB, 6MB, 8MB, 10MB} are {71.28%, 71.55%, 71.63%, 71.68%, 71.72%, 71.75%, 71.76% }[1], when the L2-Block size increases, the buffer size needed to store the decompressed L2-Block increases super-linearly. The average decompressed data sizes are then given by {3.48MB, 7.03MB, 10.57MB, 14.12MB, 21.22MB, 28.32MB, 35.41MB}, respectively.

The performance degradation is due to several reasons. Firstly, the size of the input data (the number of Mappers) is certainly affecting the overall performance. Secondly, when the L2-Block size increases, the RaPC-L2 decompression and the processes for further parsing records at each Mapper will take longer time, accordingly. Figure 6.11 (*upper-left*) shows the average time used by Mappers for the *Word Count* job. Additionally, before the intermediate data is shuffled to the corresponding Reducers, there are *Partitioning*, *Sorting* and *Combining* (optional) processes. These processes are operated on a per Map output data basis. That is, each Mapper takes a data split (with the size equal to the HDFS-Block size) and produces a group of intermediate records. The intermediate output data from each individual Mapper will have to go through the processing pipelines in order (*Partitioning*, *Sorting* and *Combining*). When the L2-Block size increases, the compression ratio increases accordingly as we have seen before. Given the fixed size of the data split, the higher compression ratio implies more data needs to be processed by each individual Mapper. This potentially increases the number of intermediate records per Mapper. With the increased number of intermediate records, the *partitioning* and *sorting* process will take longer to complete. However, considering the dynamic network activity in the cluster, for example, communications required by other services such as HDFS monitoring and orchestration components, thus the *shuffling* time varies slightly as shown in Figure 6.11 (upper-right).

From the compression algorithm perspective, when the L2-Block size is greater than 1MB, the compression ratio tends to be stable. This is a general characteristic of the RaPC-L2 scheme. An increase in L2-Block size does not have much effect on reducing the data size. Especially, when the HDFS-Block size is larger, the influence of the number of data splits (number of Mappers) becomes smaller, but it increases the workload per Mapper super-linearly and it also implies that more time is required for data preprocessing, such as forming internal data structures. In balancing MapReduce performance and resource usage, an L2-Block size from the range [512KB, 2048KB] is generally optimum.

[1]Note that, in Figure 6.13 and Figure 6.14, the compression ratio for dataset *DS-WikiEN* is shown as 67.51% for a L2-Block size of 1MB. Here we show the RaPC-L2 compression ratio is 71.28% for the same L2-Block size. This is because the former compression ratios were calculated from compressing 2GB data which was drawn from the entire dataset. In this experiment, the compression ratios are calculated from compressing the entire 47GB dataset. Due to the different data sizes and content variation, the compression ratio varies accordingly.

### 6.3.3 Further Compression

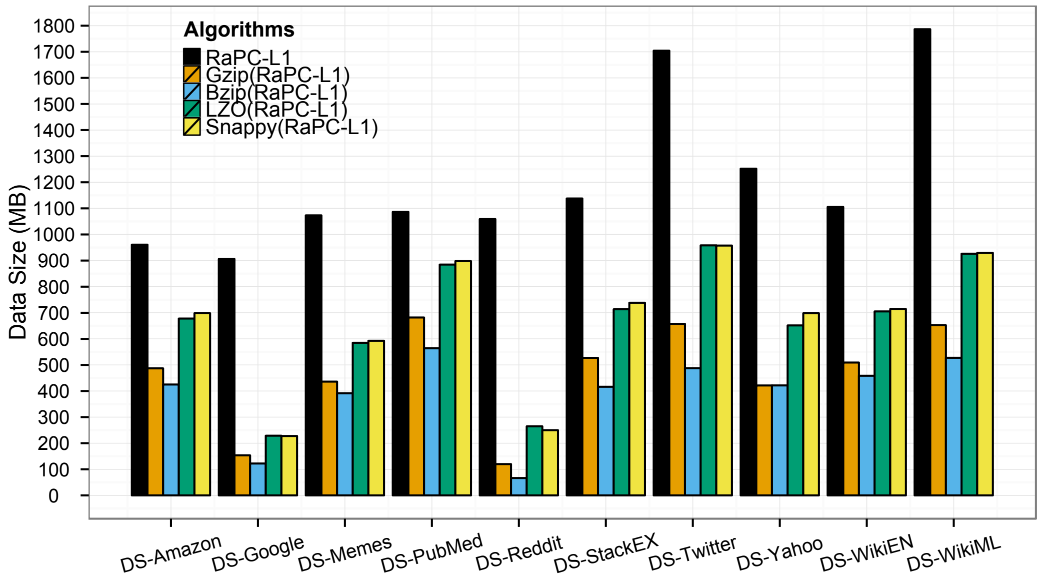


Figure 6.12: Further compression on RaPC-L1 compressed data using general purpose compressors with default compression level configured for *Bzip*, *Gzip* and *LZO*.

In order to efficiently distribute the Map output data to Reducer nodes during the *shuffling* phase, Hadoop allows us to compress the intermediate data with options of several compression algorithms, including *Gzip*, *Bzip*, *LZO*, *Snappy* and *LZ4* (recently). *Gzip* and *Bzip* offer high compression ratios; *LZO*, *Snappy* and *LZ4* are speed-optimized. The following experiments exclude the *LZ4* compressor as it has similar results to *LZO*. When using the RaPC-L2(L1) compressed data, the Mapper output data is in RaPC-L1 compressed format. Since the data has already been compressed, the original pattern of the contents is changed. This may affect the compression ratios produced by the general purpose compressors. In this experiment, we study whether it is worth applying further compression using the aforementioned compressors to the RaPC-L1 compressed data. Figure 6.12 shows the compression results from *Gzip*, *Bzip*, *LZO* and *Snappy* on RaPC-L1 compressed data. We compress exactly 2GB data from each dataset (as given in Appendix A <Table A.2>) using the RaPC-L1 compressor. The compressed data from RaPC-L1 is then fetched to these general purpose compressors. From the results shown in Figure 6.12, we observe an over 50% data size reduction for most of the datasets, thus we still can take advantage of those general compressors for the data *Shuffling* process when working with RaPC-L2(L1) compressed data.

### 6.3.4 RaPC Characteristics

In this section, we evaluate the characteristics of the RaPC-L1, -L2 and -L2(L1) schemes including compression ratio and compression/decompression speed. The results are thereafter compared with four common compressors that are currently supported by the Hadoop system, including *Gzip* (*v1.6*), *Bzip* (*v1.0.6*), *LZO* (*v1.03*) and *Snappy* (*v1.1.2*). Exactly 2GB of data is drawn from each dataset (as given in Appendix A <Table A.2>) for the experiments. There are five runs for each experiment. The compression ratio for *Gzip*, *Bzip*, *LZO* and Snappy are identical across the five runs. In contrast, there are some slight variations in the RaPC-L1, -L2 and -L2(L1) compression results due to the random sampling effects. The compression/decompression speed also varies slightly due to system variation of the testbed. The mean values and standard mean errors are calculated and included in the experimental results. All experiments are carried out on a Linux kernel version 3.19.0 and x86_64 architecture platform with dual WD5000AAKS-75V0A0 500GB hard disks (7200RPMs) and *Ext4* (*version 1.0*) file system.

#### 6.3.4.1 Compression Ratio

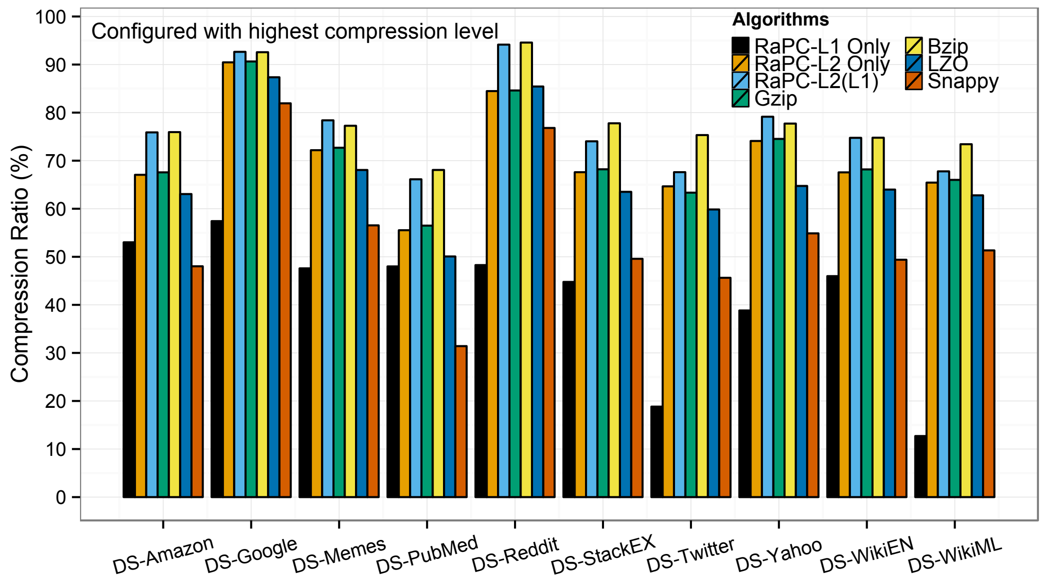


Figure 6.13: RaPC compression ratio comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

Compression ratio is content dependent. Figure 6.13 shows the compression results from RaPC, *Gzip*, *Bzip*, *LZO* and *Snappy*. Specifically, *Gzip*, *Bzip* and *LZO* are configured with the highest compression level. RaPC-L1 can compress data by ∼50% on average. There are two exceptions on dataset *DS-Twitter* and dataset *DS-WikiML*. Recall that

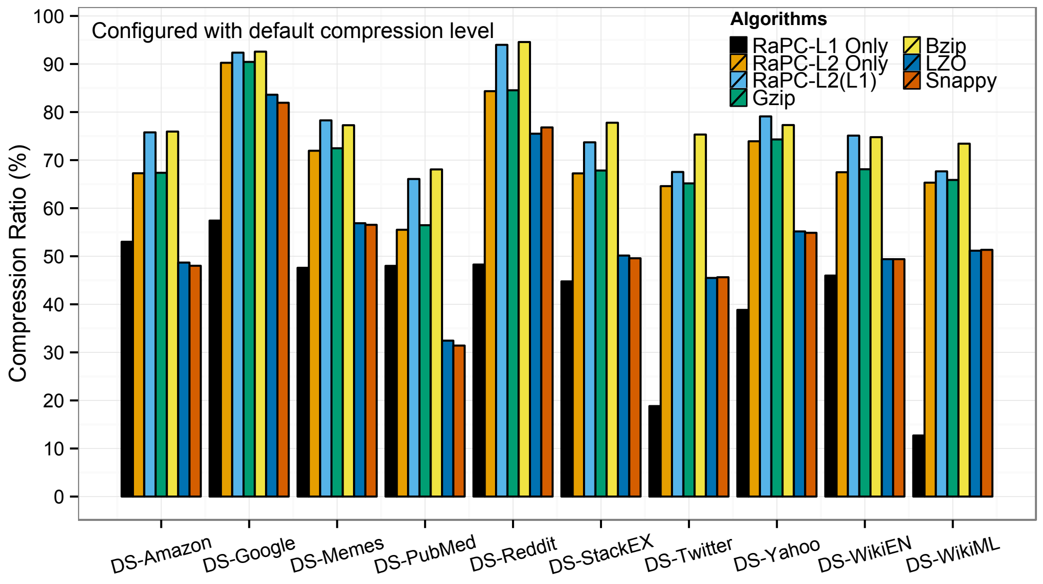


Figure 6.14: RaPC compression ratio comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

RaPC-L1 does not compress Unicode contents. Dataset *DS-WikiML* is multi-language Wikipedia articles, where the majority of the contents are complex Unicode texts which are incompressible. Additionally, we must use the *0x11* and *0x12* character pair to enclose any Unicode strings. This leads to the low RaPC-L1 compression ratio on dataset *DS-WikiML*. Dataset *DS-Twitter* contains ∼490 million Twitter *tweets* collected world wide. A large number of records contain Unicode texts. Moreover, each tweet record is comprised of time-stamp, anonymized user ID (hash codes) and tweet topics. This makes the vocabulary size for the dataset much larger. Considering that the RaPC-L1 code length increases with the vocabulary size, this makes the average code-length longer for the dataset. In practice, if the data is known to contain a lot of Unicode content, it is better to avoid using RaPC-L1 compression at all.

RaPC-L2 is a block-based compression based on the *Deflate* algorithm. In principle, it can achieve the same compression ratio as *Gzip*. Recall that in RaPC-L2 we intentionally break the connections between DO-Blocks and stuff trailing bytes at the end of each L2-Block; this makes RaPC-L2 compression slightly worse than *Gzip*. In fact, RaPC-L2 compression ratios are very close to *Gzip*.

RaPC-L2(L1) is a composite compressor. It applies the RaPC-L2 compression on the RaPC-L1 compressed data. In practice, it can achieve compression ratios close to *Bzip* and sometimes even better than *Bzip*. For example, the compression results from *Bzip* and RaPC-L2(L1) for dataset {*DS-Google*, *DS-Memes*, *DS-Twitter*, *DS-WikiEN*} are given by {{152.2MB, 150.5MB}, {465.6MB, 442.4MB}, {663.5MB, 663.3MB}, {456.3MB, 427.1MB}}, respectively.

Many modern compressors have an option to compress data at different levels. Compressing at a higher level can achieve better compression ratios. On the other hand, it will take considerably longer time. In Figure 6.13, *Bzip*, *Gzip* and *LZO* are configured with the highest compression level. Figure 6.14 shows the comparison results when configured with default compression levels. The compression ratios for RaPC-L1, -L2, -L2(L1) and *Snappy* remain unchanged; *Bzip* produced almost identical results. For *Gzip*, there is a slight drop in the compression ratio. This makes our RaPC-L2 as good as *Gzip* and even slightly better on dataset *DS-Google*. There is a big drop for *LZO* of approximately $15\%$ on average between highest and default compression levels. Note that the drops are percentages. If the original dataset is large, this drop can be significant. For example, the precise figures of the compression results from RaPC-L1, -L2, -L2(L1), *Gzip*, *Bzip* and *LZO* with the highest and default compression level on dataset *DS-Amazon* (2GB original data size) are given by {962MB, 670MB, 496MB, 664MB, 492MB, 757MB} and {962MB, 670MB, 496MB, 668MB, 492MB, 1051MB}, respectively. The *LZO* compression ratios dropped by 14.3% from 63.0% ($\frac{2048MB-757MB}{2048MB}$) to 48.7% ($\frac{2048MB-1051MB}{2048MB}$), corresponding to a 294MB size difference. If we assume that the compression ratio is constant with data size, referring to Appendix A (Table A.2), the total data size of dataset *DS-Amazon* is 33.4GB, with the best and default compression level configured for *LZO*, the compressed size is $\sim$21GB and $\sim$16GB, respectively, giving a 5GB difference. The comparison results are not intended to argue against the use of *LZO* compression. But, they reveal whether the compression algorithms are consistent across data contents. It is important for RaPC to use a relatively consistent algorithm for the RaPC-L2 compression.

#### 6.3.4.2 Compression Speed

For compression speed, the current implementation of RaPC-L2 achieves similar speed to *Gzip* and RaPC-L2(L1) is similar to *Bzip*. Figure 6.13 and Figure 6.14 show the comparison results for the highest and default compression levels configured for *Gzip*, *Bzip* and *LZO*, respectively. In general, *Bzip* is the slowest compressor because it has an extra step for block sorting. The speed is often influenced by block size. Interestingly, when *LZO* is configured with the highest compression level, it takes much longer than all other compressors as shown in Figure 6.15, but it still remains the fastest on decompression. In contrast, RaPC-L2 and *Gzip* are relatively stable and consistent. Figure 6.17 and 6.18 shows the decompression speed for each compressor configured with the highest and default compression levels. Comparing the two figures, the results are almost identical. There are only tiny differences due to the compressed data sizes being slightly different. Thus, the decompression speed is not correlated with the compression levels.

In designing RaPC, we aimed to reduce the data size as much as possible. This is driven

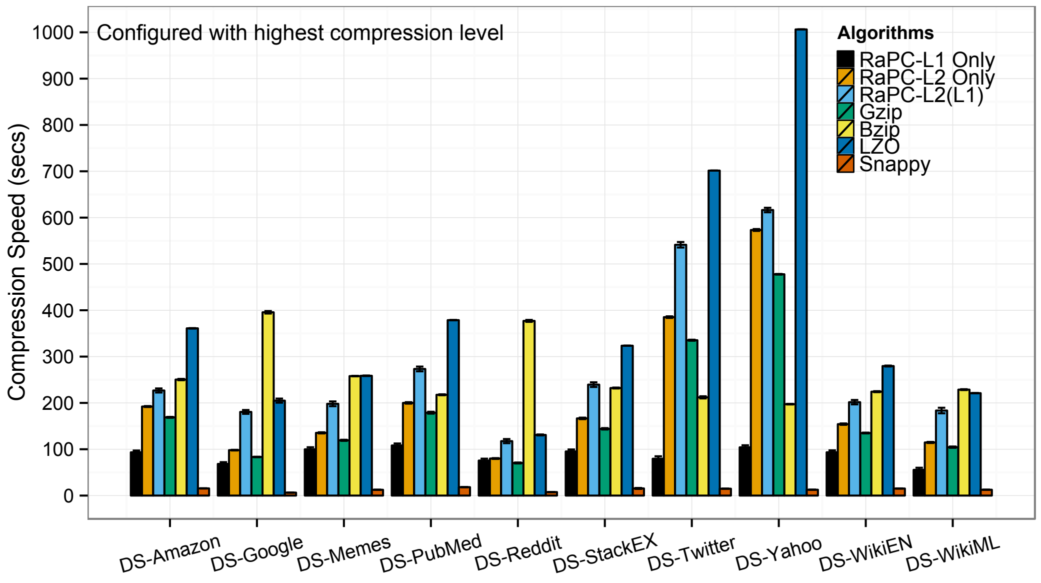


Figure 6.15: RaPC compression speed comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

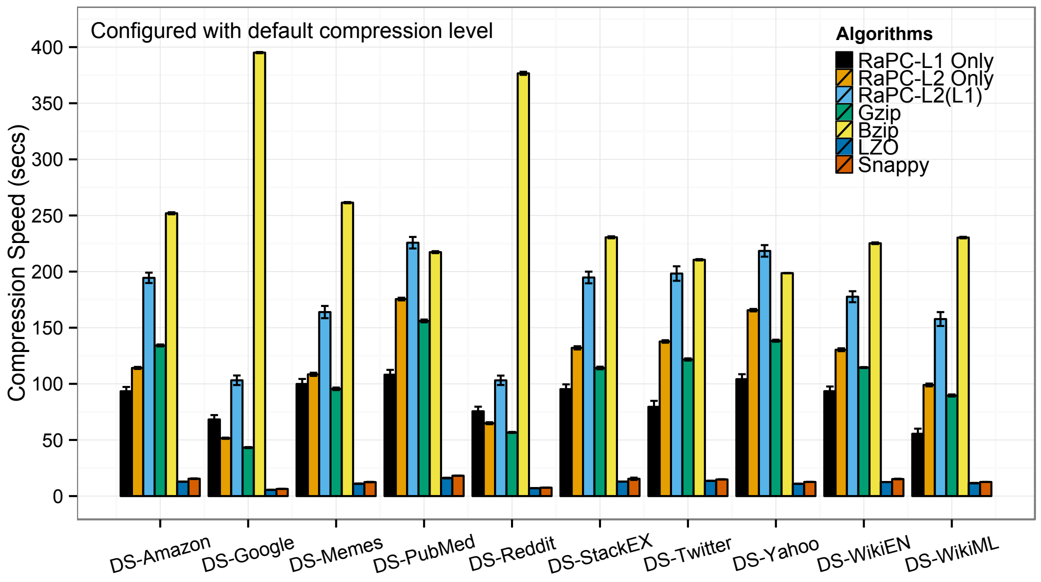


Figure 6.16: RaPC compression speed comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

by the fact that data size is the predominant factor affecting the overall performance of MapReduce programs. Also, from the previous chapter, the evaluation results from comparing RaC-Deflate and RaC-LZ4 show that the decompression time is insignificant compared to the time used for loading data from hard disks to memory. RaPC Layer-2 needs to be a scheme that is best balanced between compression ratio and speed. *Gzip* is the best candidate for RaPC-L2 and it is based on the *Deflate* algorithm.

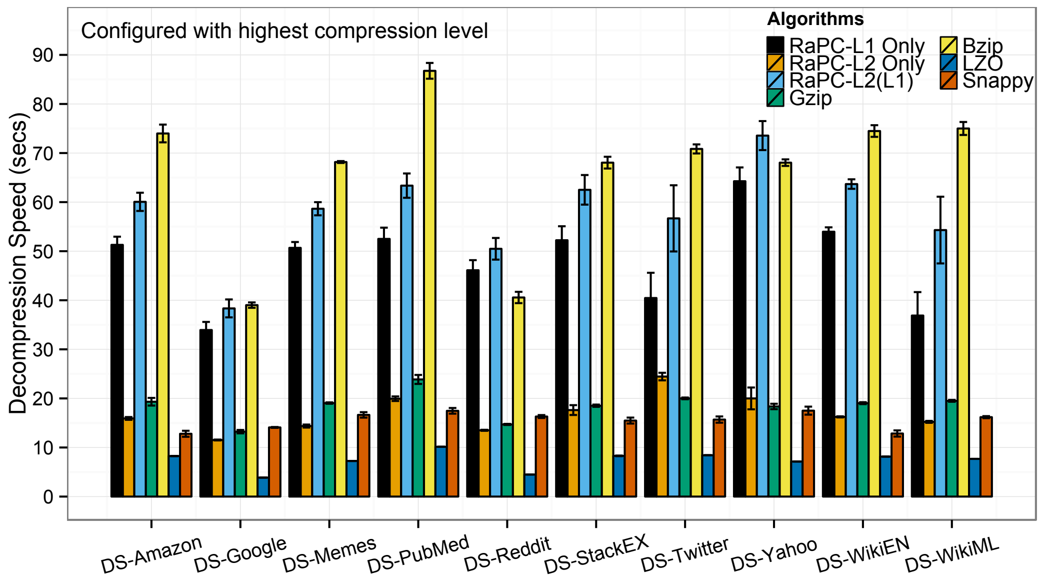


Figure 6.17: RaPC decompression speed comparing to Hadoop supported algorithms with highest compression level configured for *Bzip*, *Gzip* and *LZO*.

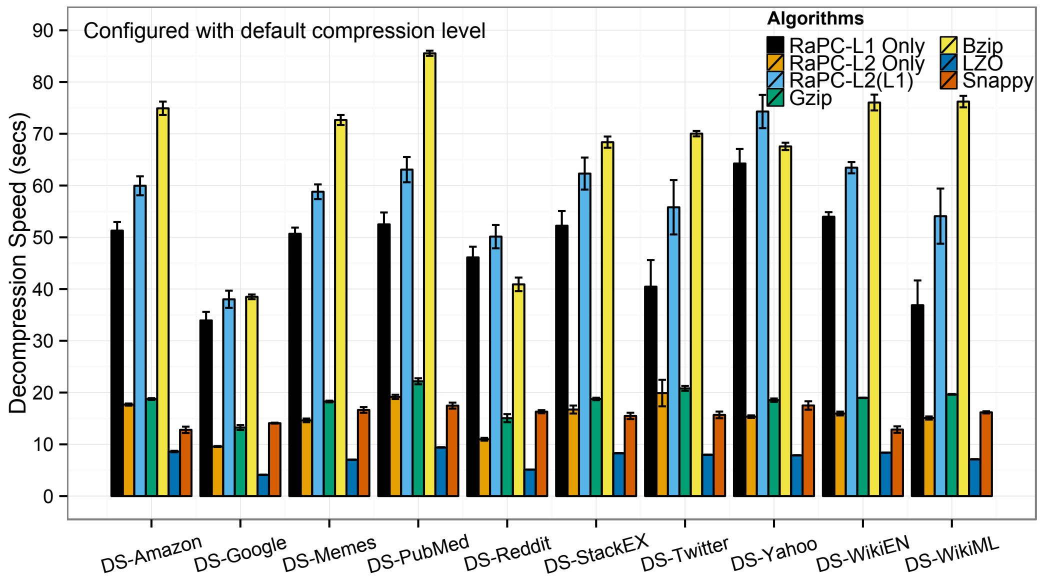


Figure 6.18: RaPC decompression speed comparing to Hadoop supported algorithms with default compression level configured for *Bzip*, *Gzip* and *LZO*.

## 6.4 Conclusion

For big data, analytic platforms are mainly challenged by data volumes. Reducing data size without affecting its integrity and usability is our overall objective in mitigating the big data issue. In this chapter, we presented the RaPC, a new two-layer architecture for textual data compression. Each layer of compression is designed to improve the performance of the corresponding stage of data processing in MapReduce. Generally, RaPC Layer-1 can achieve compression ratios similar to *LZO* and *Snappy*, RaPC Layer-2 is close to *Gzip*, and RaPC-L2(L1) is comparable with the *Bzip* compressor.

The effectiveness and flexibility of RaPC was demonstrated through various standard MapReduce jobs with a set of real-world datasets having different properties. Our experimental results have shown that reducing data size can effectively improve analysis performance and make corresponding efficient use of resources. Furthermore, we studied the relationships between the workload per Mapper, number of Mappers per job, MapReduce analysis performance and resource utilization. Our analysis results indicate that the RaPC Layer-2 block size is the single parameter that most affects the balance between performance, memory usage and compression ratio. Finally, RaPC is a platform independent solution. It can be used with other analysis frameworks (e.g., Spark and R) with supporting libraries for each framework.

The general idea is therefore if algorithms can be tuned for particular tasks, data can also be optimized for improving data processing and analysis efficiency at different stages.

# Chapter 7

# Conclusion

*"That which is achieved the most, still has the whole of its future yet to be achieved."*
— Lao Tzu

The sheer size of the data is a major challenge to big data analytics. The use of compression is an attractive approach to mitigate the big *volume* issue. However, existing compression schemes do not have the desired properties for allowing the compressed data to directly participate in big data analysis in a seamless fashion.

This research set out with the goal of investigating novel compression schemes for mitigating big data issues. In seeking solutions, we analyzed the data organization in the Hadoop Distributed File System (HDFS) and data processing in the MapReduce computational framework. The outcome of this analysis is a set of requirements that must be satisfied by a potential compression scheme. We then conducted a literature review on conventional and emerging compression schemes in order to identify their suitability and potential. We summarized our findings: that processing compressed data in parallel in a distributed environment firstly requires the compressed data to be splittable; that a context-free scheme is an appropriate method that allows the compressed data to be directly processable; that separating compression models and compressed contents is necessary for providing transparency between the compressed data and MapReduce developers; that being content-aware allows the seamless integration of compression, HDFS and MapReduce. The resulting outputs of this research are the Content-aware Compression (CaC) schemes. We evaluated CaC with a set of standard MapReduce jobs using a collection of real-world datasets. The evaluations show very encouraging results when compared to standard analysis methods.

In this chapter, we begin by summarizing our contributions, followed by possible future work, finally discussing implications of the research.

## 7.1 Contributions

Leveraging novel compression schemes for improving performance and reducing resource requirements for big data analysis is our overall contribution to the field of big data research. In this thesis, we presented several compression schemes having different properties suitable for a variety of big data analysis scenarios. This section is organized in such a way that we map the research questions set up in Chapter 1 to our contributions.

> **Research Question**: Whether we can compress data in such a way with regards to the characteristics of big data, so that the compressed data can be directly processed in MapReduce without decompression, while ensuring the compressed data is compatible with existing algorithms and software packages as well as transparent to MapReduce program developers.

**Our first contribution** is the Content-aware Partial Compression (CaPC) scheme presented in Chapter 3. The CaPC design was based on several observations. Firstly, textual data produced from a single source (e.g., social media or a machine log) generally shares a common vocabulary. Thus, the vocabulary can be used as a compression model. For a small dataset, the compression model can be as big as the original dataset. It may not offer any advantage for compression. However, as the data size increases, the size of the compression model remains relatively constant, and then better compression is achieved. Secondly, data is being produced from a great variety of sources. Different data sources organize information in various formats. There is a wealth of available software packages and algorithms developed for efficiently parsing and processing data in specific formats. In order to make the compressed data transparent to the existing software and algorithms, we compress data selectively in which informational content is compressed and functional content is left intact. Thirdly, the current version of MapReduce only supports limited string (text data) encoding schemes. Compression should not break the rules used by standard encoding schemes. As a unique feature, we use specially designed codes that are valid strings rather than simply binary sequences. Lastly, the separation of compression model(s) and compressed contents along with the provided supporting library, together make the use of CaPC compressed data transparent to MapReduce developers. CaPC was implemented for English text in this work. It can be also used for compressing other languages by building the static compression models for each language.

**Our second contribution** is the Approximated Huffman Compression (AHC) which is designed for domain-specific data, presented in Chapter 4. AHC is a bit-oriented, character-based compression scheme. It thus supports arbitrary pattern searching. We use a hybrid data structure that combines a lookup table and a Huffman tree to achieve data decompression in constant time. We also provide algorithm analysis on AHC performance in terms of compression ratio and decompression speed. As an addition, AHC generates a very small (approximately 2KB) compression model per compressed file. The compression model can then be used as a security key for protecting data in a public environment, when encryption is not necessary but some privacy is useful. Additionally, as AHC is a character based compressor, it supports compressing texts in multiple languages.

Our first and second contributions are based on specially designed context-free schemes which balance compression ratio and direct processability (without decompression). Additionally, if a MapReduce program can perform analysis with compressed data without decompression, the Mappers will also produce intermediate data in the compressed format. Compressing the intermediate data using general purpose compressors can further reduce the data size, thus improving efficiency for data transmission over the network.

> **Research Question**: Whether, when data is compressed by modern compression schemes such as *Gzip* or *LZO*, it is possible to perform decompression in memory as data is being consumed at each computational node of Hadoop (parallel decompression in memory) so that the total decompression time can be reduced in proportion to the number of parallel processes, the data loading time can be kept to a minimum and the use of extra storage space can be avoided.

**Our third contribution** explores using modern compressors to achieve a high level of compression without considering direct processability. That provides a more generic solution. Whereas modern processor and memory speed have improved greatly, there has been little progress on improving I/O performance (especially the I/O system between commodity hard disks and memory which still presents a bottleneck in *Von Neumann* architectures). Therefore, the adoption of the procedures of reading compressed data from hard disks to memory, then decompressing it in memory has the potential (decompression time also needs to be taken into consideration) advantage of better performance over reading original data straight into memory. This has been shown by *LZO*-splittable developed by other practitioners [KML13] and further proved by our Record-aware Compression (RaC) schemes. However, *LZO*-splittable and similar approaches perform blind compression on data which often requires additional auxiliary indexing files or a separate system to make compressed data splittable to HDFS. This subsequently leads to data locality preservation issues as discussed in Chapter 5. In contrast, the RaC scheme is designed to overcome these issues by making compression content-aware at the level of logical records of a dataset, controlling the use of contextual data, and using an efficient packaging mechanism that avoids the need for indexing or dedicated systems without sacrificing much compressibility.

Moreover, as compression algorithms often have to achieve a balance between compression ratio and decompression speed, we investigated whether a scheme with higher compression ratio and lower decompression speed can achieve better overall performance than one with a lower compression ratio and higher decompression speed. This has not been addressed previously.

**Our fourth contribution** lies in the conclusion that a higher compression ratio with relatively slower decompression speed can achieve better overall performance on commodity servers. This conclusion is based on comparing two implementations of the RaC scheme, namely RaC-Deflate and RaC-LZ4, in the environment as given in Appendix A.

> **Research Question**: Whether we can combine the CaPC/AHC and RaC approaches to take advantage of both.

**Our fifth contribution** lies in synthesizing our previous findings. We note that a MapReduce program benefits from the CaPC and AHC at the data processing stage, whereas RaC improves performance at the data loading stage. We designed a layered architecture (RaPC) so that the MapReduce process can at different stages take advantage of the corresponding layers.

RaPC consists of two compression schemes. The first scheme (RaPC Layer-1) is byte-oriented context-free compression. We developed a dynamic coding scheme in which the codeword length increases along with the compression according to the variety of information to be encoded. RaPC Layer-1 compression follows the same principles used in CaPC and AHC. The output of RaPC Layer-1 is directly processable data and achieves a data size reduction of approximately 50%. In other words, processing RaPC Layer-1 compressed data in Hadoop requires half of the system memory used when processing data in the original format. The RaPC Layer-2 scheme is based on a modified *Deflate* compression algorithm. The scheme follows the principles used by RaC. We also developed a more compact version of the packaging mechanism compared to that used by RaC, for an improved compression ratio. RaPC works by firstly compressing data using the Layer-1 scheme and the compressed data is then compressed by the Layer-2 scheme. As a result, we can achieve data compression ratios close to the *Bzip* compressor. In MapReduce, the Layer-2 compressed data is unpacked during the data loading phase; the Layer-1 compressed data is directly used for processing/analysis. Evaluations based on several standard MapReduce jobs with a set of real-world datasets show that using RaPC can achieve substantial improvement on analysis performance, up to 72%. Additionally, the RaPC-L2 schemes can potentially be used for other types of data, such as images, audio and video, as long as the data splitting boundaries can be identified.

In this thesis, We have presented several novel compression schemes with concrete implementations coping with different scenarios for textual data analysis on Hadoop. The CaC schemes proposed here are in fact full solutions for Hadoop and MapReduce to work with compressed data rather than once-off compression schemes. The CaC schemes improve performance for various aspects of data analysis including analysis efficiency, cluster memory usage, storage requirements and cost of transmitting data over the network. The evaluations of CaC schemes were conducted using standard analysis jobs (written in MapReduce), for a set of well-known real-world datasets as listed in Appendix A (Table A.2), in a fully operational Hadoop cluster as shown in Appendix A (Figure A.1), with detailed configurations as listed in Appendix A (Table A.1). Thus, our experimental environment is replicable. Examples of CaC schemes (AHC and RaPC) are given in Appendix C. As illustrated, they have minimum impact on writing MapReduce programs. Additionally, introducing CaC schemes to other analytic platforms, such as Spark, only requires extra supporting utility functions without modification to the underlying system. As a result, there is no barrier to industry adoption of CaC schemes and the value of using CaC schemes can be realized immediately.

## 7.2 Future Directions

The general idea of CaC schemes can be adopted by any big data analytic platform. In this work, we conducted evaluation and implemented supporting libraries primarily for Apache Hadoop. As CaC schemes are presented as systematic solutions, possible future work using the same idea are apparent. Moving forward from the current implementation, future directions involve extending CaC schemes to:

1. *other analytic platforms*. In this work, we evaluated CaC schemes on the HDFS file system of Hadoop. In principle, CaC schemes are agnostic to the underlying distributed file systems and storage. It may also be worth investigating how CaC schemes can be beneficial to Hadoop based on other storage systems such as GFS (Google File System) [GGL03], QFS (Quantcast File System) [ORR$^+$13] and NoSQL (Not Only Structured Query Language) [Cat11] databases.

   *A computation framework* is another important component of an analytic platform. Apart from MapReduce, Spark is an increasingly popular framework which focuses on in-memory processing and supporting Directed Acyclic Graph (DAG) work-flows. As memory is still considered to be an expensive hardware resource (comparing to commodity hard disks), using CaPC or RaPC Layer-1 compression can greatly reduce memory consumption, since the CaPC or RaPC Layer-1 compressed data can be processed directly without decompression. We will extend CaC schemes to work with Spark in future work.

2. *other data processing paradigms*. Currently, *batch* and *stream* are the two main data processing paradigms for big data. In this work, we primarily focused on *batch* processing where data has already been collected and is thereafter analyzed/processed offline. In contrast, *stream* processing presents different charac-

teristics. It requires data to be collected, analyzed and acted upon in real-time. To improve performance, data can be compressed using the lightweight CaPC or RaPC Layer-1 scheme at source (this is especially suitable for sensors or devices that are not capable of carrying out complex computations), then sent to analytic platforms for processing in compressed format.

Other future work involves investigating application-specific data in which the data contents do not vary much (examples like machine log files and sensor data), so that static compression models are appropriate. For these cases, compression models can be built into the analysis programs, statically. This can improve overall analysis performance by avoiding the need to load the compression models on the fly. It is also possible for stream processing where compressed data is being received.

## 7.3 Epilogue

The real value of big data has been gradually realized and boosted recently by successes in both public and private sectors. This encourages decision makers to accumulate more data and perform analysis at an increasingly larger scale. To what extent this can be done, perhaps is a hard question. As the ever-increasing volume of data is continuously challenging data analysis tools, algorithms and platforms, data compression is an effective way of reducing data size. However, existing compression schemes do not offer significant benefits to big data analysis, as already discussed in this thesis. In order to integrate data compression and big data analysis, designing new compression schemes that take into consideration the characteristics of data content, computational model and analytic platform is necessary. In response, we have developed the CaC schemes.

The CaC schemes have been implemented as full solutions supporting ease of use by developers and orthogonal to other possible optimizations. The advantages of the CaC schemes for textual data analysis have been demonstrated using a variety of standard real-world benchmarks. As well as the existing benefits, the CaC schemes also provide the basis for future extensions to other types of data such as audio and video, and for future applications in other analytic platforms.

# Appendix A

# Experimental Environment

The hardware specification (Table A.1) and cluster topology (Figure A.1) remain the same for the evaluations carried out for the AHC, RaC and RaPC schemes. Cluster nodes are connected to a NetGear GS716T Gigabit Ethernet switch. In total, there are eleven datasets (Table A.2) used for the evaluation of the proposed CaC schemes introduced in Chapter 3 4 5 and 6. Additionally, 2GB of data from each dataset is used for evaluating the performance of CaC schemes in terms of compression ratio, compression speed and decompression speed. Some of the datasets may not be directly accessible. Contact the corresponding organizers to obtain an access link.

Table A.1: Experimental environment: hardware specification.

| Node | CPU | Memory | Storage | Network |
|---|---|---|---|---|
| NameNode | Intel Core i5 (2415M) 2.30 GHz | DDR3 8 GB | WD7500BTKT 750 GB | BCM57765 Gigabit NIC |
| DataNode-1 | Intel Core 2 Duo (E8400) 3.00 GHz | DDR2 8 GB | WD10EZRX 1.0 TB | Intel 82567LM-3 Gigabit NIC |
| DataNode-2 | Intel Core 2 Duo (E8400) 3.00 GHz | DDR2 8 GB | WD10EZRX 1.0 TB | Intel 82567LM-3 Gigabit NIC |
| DataNode-3 | Intel Core 2 Duo (E8400) 3.00 GHz | DDR2 8 GB | WD10EZRX 1.0 TB | Intel 82567LM-3 Gigabit NIC |
| DataNode-4 | Intel Core 2 Duo (E8400) 3.00 GHz | DDR2 8 GB | WD10EZRX 1.0 TB | Intel 82567LM-3 Gigabit NIC |
| DataNode-5 | Intel Core 2 Duo (E8400) 3.00 GHz | DDR2 8 GB | WD10EZRX 1.0 TB | Intel 82567LM-3 Gigabit NIC |
| Management | Intel Core 2 Duo (E6550) 2.33 GHz | DDR2 8 GB | WD5000AAKS 500 GB | Intel 82566DM-2 Gigabit NIC |

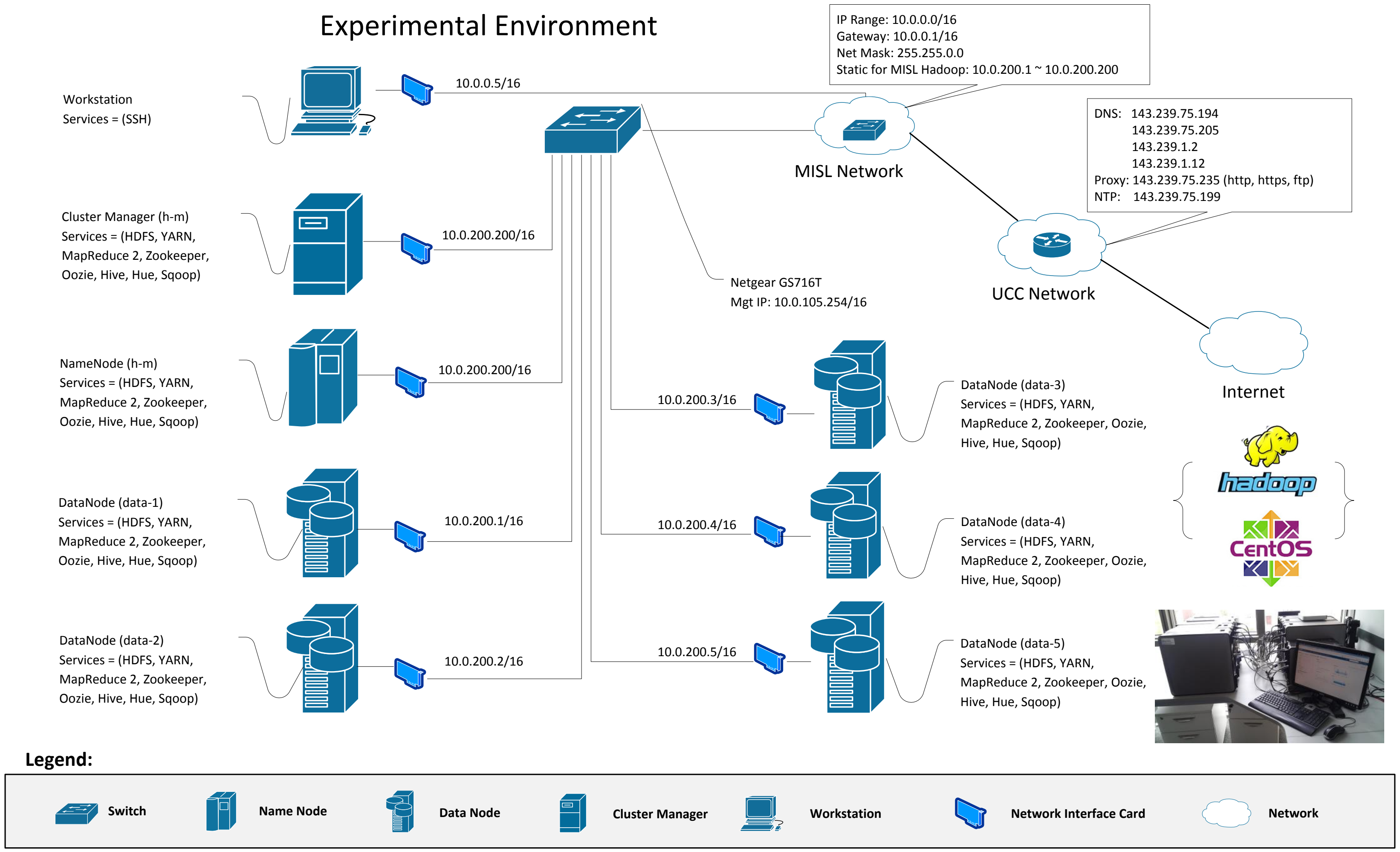

Experimental Environment
IP Range: 10.0.0.0/16
Gateway: 10.0.0.1/16
Net Mask: 255.255.0.0
Static for MISL Hadoop: 10.0.200.1 ~ 10.0.200.200
Workstation
Services = (SSH)
10.0.0.5/16
MISL Network
DNS: 143.239.75.194
143.239.75.205
143.239.1.2
143.239.1.12
Proxy: 143.239.75.235 (http, https, ftp)
NTP: 143.239.75.199
Cluster Manager (h-m)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
10.0.200.200/16
UCC Network
Netgear GS716T
Mgt IP: 10.0.105.254/16
NameNode (h-m)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
10.0.200.200/16
Internet
10.0.200.3/16
DataNode (data-3)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
DataNode (data-1)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
10.0.200.1/16
10.0.200.4/16
DataNode (data-4)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
hadoop
CentOS
DataNode (data-2)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
10.0.200.2/16
10.0.200.5/16
DataNode (data-5)
Services = (HDFS, YARN, MapReduce 2, Zookeeper, Oozie, Hive, Hue, Sqoop)
Legend:
Switch
Name Node
Data Node
Cluster Manager
Workstation
Network Interface Card
Network

Figure A.1: Experimental environment: Hadoop cluster topology

Table A.2: List of public and private datasets used for evaluating the Content-aware compression schemes.

| Index | Dataset and Source | Format | Size (GB) | Description |
|---|---|---|---|---|
| DS-Amazon | Amazon Product Reviews* [ML13] | TXT | 33.4 | The dataset contains ∼35 million reviews on ∼2.5 million products from ∼6.6 million users. |
| DS-Google | Google Server Logs [CRH11] | CSV | 158.9 | The dataset contains ∼1.2 billion server usage traces from a Google cluster (US Eastern) of ∼11 thousand machines. |
| DS-Memes | Memetracker Memes* [LBK09] | TXT | 52.5 | The dataset contains ∼96 million documents; ∼211 million memes; and over 418 million URL links from Memetracker. |
| DS-PubMed | PubMed Records [LAB+09] | CSV | 3.1 | The dataset contains ∼22 million publication records of ∼11 million authors from PubMed database. |
| DS-Reddit | Reddit Submissions* [LML13] | HTML | 13.0 | The dataset contains ∼132 thousand submissions from reddit.com. |
| DS-StackEX | StackExchange Posts [Int] | XML | 40.6 | The dataset contains an anonymized dump of user-contributed contents on the Stack Exchange network. |
| DS-Twitter | Tweet Topics [WM14] | TSV | 17.3 | The dataset contains ∼490 million tweet records. Each record consists of *timesamp*, *anonymized user_id*, and *topics*. |
| DS-Yahoo | Yahoo! Music Ratings* [Yah06] | TSV | 10.2 | The dataset contains ∼700 million ratings on ∼136 thousand songs from ∼1.8 million users of Yahoo! Music service. |
| DS-WikiEN | Wikipedia Article Abstract [Wika] (English) | XML | 47.0 | The dataset contains the latest article abstracts (English) from Wikipedia (09-December-2014). |
| DS-WikiML | Wikipedia Article Abstract [Wikb] (Multi-Language) | XML | 4.3 | The dataset contains the latest article abstracts (Multi-language) from Wikipedia (09-December-2014). |
| DS-mRNA | Human mRNA Sequence [Con09] | TXT | 12.5 | GenBank mRNAs from the part of human genome (hg19, GRCh37 Genome Reference Consortium Human Reference 37 <GCA_000001405.1>). |

*: Dataset may not be directly accessible. Contact corresponding sources to retrieve download links.
*1GB = 1,073,741,824 Bytes*

# Appendix B

# AHC Codeword Example

Table B.1: Codeword generated for dataset *DS-WikiEN* using the AHC compression.

| Hex | Frequency | Codewords | Hex | Frequency | Codewords |
|---|---|---|---|---|---|
| 0x20 | 1654020 | 101 | 0xB9 | 837 | 11000001110110 |
| 0x6E | 557901 | 0001 | 0x9D | 864 | 11000001110111 |
| 0x6F | 591786 | 0010 | 0xB8 | 891 | 11000001111001 |
| 0x69 | 601101 | 0011 | 0xA9 | 918 | 11000001111010 |
| 0x61 | 634338 | 0101 | 0x58 | 945 | 11000001111111 |
| 0x74 | 745767 | 1000 | 0x82 | 999 | 11101101111011 |
| 0x65 | 934983 | 1101 | 0x9B | 270 | 010000011100010 |
| 0x64 | 264843 | 00001 | 0xA3 | 270 | 010000011100011 |
| 0x68 | 312903 | 01001 | 0xB4 | 270 | 010000011100100 |
| 0x6C | 366390 | 01111 | 0xCB | 270 | 010000011100101 |
| 0x73 | 494559 | 11100 | 0x85 | 297 | 010000011111110 |
| 0x72 | 517185 | 11111 | 0xAA | 297 | 010000011111111 |
| 0x79 | 127818 | 000000 | 0xB7 | 297 | 011010001001100 |
| 0x66 | 180846 | 011100 | 0xAB | 324 | 011010001001101 |
| 0x67 | 182223 | 011101 | 0x86 | 351 | 100101111000110 |
| 0x70 | 186840 | 100100 | 0xB0 | 351 | 100101111000111 |
| 0x6D | 209763 | 100111 | 0xAF | 378 | 100101111011110 |
| 0x75 | 253611 | 111100 | 0xA6 | 432 | 110000011110000 |
| 0x63 | 261063 | 111101 | 0xB3 | 459 | 110000011110110 |
| 0x27 | 68418 | 0000011 | 0xC5 | 459 | 110000011110111 |
| 0x76 | 75924 | 0100011 | 0xC9 | 459 | 110000011111100 |
| 0x2F | 77031 | 0110000 | 0x88 | 486 | 111011011110100 |
| 0x30 | 81864 | 0110011 | 0xBB | 486 | 111011011110101 |
| 0x31 | 85455 | 0110101 | 0xBE | 513 | 111011011111001 |
| 0x2C | 89586 | 0110110 | 0xBF | 513 | 111011011111010 |
| 0x26 | 94149 | 1001010 | 0xA2 | 540 | 111011011111100 |
| 0x3B | 99603 | 1001100 | 0xA5 | 540 | 111011011111101 |
| 0x0A | 107136 | 1100001 | 0xB5 | 540 | 111011011111110 |
| 0x7C | 108162 | 1100010 | 0xBA | 540 | 111011011111111 |
| 0x3D | 109269 | 1100011 | 0xB6 | 135 | 0100000111001100 |

| | | | | | |
|---|---|---|---|---|---|
| 0x62 | 111240 | 1100100 | 0xCC | 135 | 0100000111001101 |
| 0x2E | 115992 | 1100110 | 0xE5 | 135 | 0100000111001110 |
| 0x5D | 123768 | 1110100 | 0x5C | 162 | 0110100010110100 |
| 0x5B | 124146 | 1110101 | 0x87 | 162 | 0110100010110101 |
| 0x77 | 125226 | 1110111 | 0x8E | 162 | 0110100010110110 |
| 0x7B | 33669 | 00000101 | 0x97 | 162 | 0110100010110111 |
| 0x7D | 33696 | 01000000 | 0xA0 | 162 | 1001011110000000 |
| 0x71 | 36828 | 01000011 | 0xC4 | 162 | 1001011110000001 |
| 0x39 | 37422 | 01000100 | 0x90 | 189 | 1001011110111110 |
| 0x54 | 39744 | 01100101 | 0x92 | 189 | 1001011110111111 |
| 0x53 | 43119 | 01101001 | 0xA8 | 189 | 1100000110011000 |
| 0x41 | 45522 | 01101111 | 0xAE | 189 | 1100000110011001 |
| 0x43 | 46791 | 10010110 | 0xE3 | 189 | 1100000110011010 |
| 0x6B | 50787 | 11000000 | 0x98 | 216 | 1100000110011011 |
| 0x32 | 61020 | 11001111 | 0xAC | 216 | 1100000111100010 |
| 0x2D | 61209 | 11101100 | 0xAD | 216 | 1100000111100011 |
| 0x45 | 16497 | 000001000 | 0x8A | 243 | 1100000111111011 |
| 0x4E | 17874 | 010000010 | 0x91 | 243 | 1110110111110000 |
| 0x44 | 18198 | 010000100 | 0x95 | 243 | 1110110111110001 |
| 0x57 | 18306 | 010000101 | 0x8C | 270 | 1110110111110111 |
| 0x4C | 19035 | 010001010 | 0x40 | 81 | 01000001110011111 |
| 0x46 | 19359 | 010001011 | 0x8B | 81 | 10010111100000100 |
| 0x2A | 19494 | 011000100 | 0x9A | 81 | 10010111100000101 |
| 0x28 | 19656 | 011000101 | 0x9F | 81 | 10010111100000110 |
| 0x29 | 19656 | 011000110 | 0xE4 | 81 | 10010111100000111 |
| 0x48 | 19710 | 011000111 | 0x5E | 108 | 11000001111110100 |
| 0x52 | 19737 | 011001000 | 0x8F | 108 | 11000001111110101 |
| 0x3E | 19899 | 011001001 | 0xE6 | 108 | 11101101111101100 |
| 0x3C | 19980 | 011010000 | 0x89 | 135 | 11101101111101101 |
| 0x78 | 21978 | 011011100 | 0xFC | 1 | 0100000111001111000000 |
| 0x42 | 23544 | 100101110 | 0xFD | 1 | 0100000111001111000001 |
| 0x49 | 24786 | 100110100 | 0xFE | 1 | 0100000111001111000010 |
| 0x36 | 25299 | 100110101 | 0xFF | 1 | 0100000111001111000011 |
| 0x37 | 25299 | 100110110 | 0x01 | 1 | 01000001110011110001000 |
| 0x5F | 25299 | 100110111 | 0x02 | 1 | 01000001110011110001001 |
| 0x3A | 26838 | 110000010 | 0x03 | 1 | 01000001110011110001010 |
| 0x38 | 27324 | 110010100 | 0x09 | 1 | 01000001110011110001011 |
| 0x34 | 28107 | 110010101 | 0x0B | 1 | 01000001110011110001100 |
| 0x4D | 28350 | 110010110 | 0x0C | 1 | 01000001110011110001101 |
| 0x33 | 29430 | 110011100 | 0x0D | 1 | 01000001110011110001110 |
| 0x35 | 29484 | 110011101 | 0x10 | 1 | 01000001110011110001111 |
| 0x50 | 31401 | 111011010 | 0x1C | 1 | 01000001110011110010000 |
| 0xE2 | 8370 | 0000010010 | 0x1D | 1 | 01000001110011110010001 |
| 0x56 | 8640 | 0000010011 | 0x1E | 1 | 01000001110011110010010 |
| 0x80 | 8802 | 0100000110 | 0x1F | 1 | 01000001110011110010011 |
| 0x7A | 10422 | 0110100011 | 0x60 | 1 | 01000001110011110010100 |
| 0x6A | 11070 | 0110111010 | 0x7F | 1 | 01000001110011110010101 |
| 0x4F | 11367 | 0110111011 | 0x96 | 1 | 01000001110011110010110 |
| 0x55 | 12339 | 1001011111 | 0x9E | 1 | 01000001110011110010111 |
| 0x4A | 14094 | 1100101110 | 0xC0 | 1 | 01000001110011110011000 |

| | | | | | |
|---|---|---|---|---|---|
| 0x47 | 15444 | 1110110110 | 0xC1 | 1 | 01000001110011110011001 |
| 0x25 | 7371 | 11001011110 | 0xC6 | 1 | 01000001110011110011010 |
| 0x4B | 8046 | 11101101110 | 0xC7 | 1 | 01000001110011110011011 |
| 0x23 | 2322 | 010000011110 | 0xC8 | 1 | 01000001110011110011100 |
| 0x3F | 2619 | 011010001010 | 0xCA | 1 | 01000001110011110011101 |
| 0xC3 | 2943 | 100101111010 | 0xCD | 1 | 01000001110011110011110 |
| 0x59 | 3294 | 110000011010 | 0xD2 | 1 | 01000001110011110011111 |
| 0xD0 | 3294 | 110000011011 | 0xD3 | 1 | 01000001110011110100000 |
| 0xCE | 3321 | 110000011100 | 0xD4 | 1 | 01000001110011110100001 |
| 0x21 | 3834 | 110010111110 | 0xD5 | 1 | 01000001110011110100010 |
| 0x93 | 3834 | 110010111111 | 0xD6 | 1 | 01000001110011110100011 |
| 0xD9 | 1134 | 0100000111010 | 0xD7 | 1 | 01000001110011110100100 |
| 0x24 | 1188 | 0100000111110 | 0xDA | 1 | 01000001110011110100101 |
| 0x99 | 1215 | 0110100010000 | 0xDB | 1 | 01000001110011110100110 |
| 0xD1 | 1242 | 0110100010001 | 0xDC | 1 | 01000001110011110100111 |
| 0xD8 | 1323 | 0110100010111 | 0xDD | 1 | 01000001110011110101000 |
| 0xA4 | 1431 | 1001011110010 | 0xDE | 1 | 01000001110011110101001 |
| 0x22 | 1512 | 1001011110110 | 0xDF | 1 | 01000001110011110101010 |
| 0x2B | 1539 | 1100000110000 | 0xE7 | 1 | 01000001110011110101011 |
| 0x94 | 1539 | 1100000110001 | 0xE8 | 1 | 01000001110011110101100 |
| 0xE0 | 1566 | 1100000110010 | 0xE9 | 1 | 01000001110011110101101 |
| 0xCF | 1674 | 1100000111010 | 0xEA | 1 | 01000001110011110101110 |
| 0x51 | 1863 | 1100000111110 | 0xEB | 1 | 01000001110011110101111 |
| 0x5A | 1917 | 1110110111100 | 0xEC | 1 | 01000001110011110110000 |
| 0xBC | 540 | 01000001110000 | 0xED | 1 | 01000001110011110110001 |
| 0x83 | 567 | 01000001110110 | 0xEE | 1 | 01000001110011110110010 |
| 0xA1 | 594 | 01000001110111 | 0xEF | 1 | 01000001110011110110011 |
| 0xB2 | 594 | 01000001111110 | 0xF0 | 1 | 01000001110011110110100 |
| 0x84 | 621 | 01101000100100 | 0xF1 | 1 | 01000001110011110110101 |
| 0xA7 | 621 | 01101000100101 | 0xF2 | 1 | 01000001110011110110110 |
| 0xBD | 648 | 01101000100111 | 0xF3 | 1 | 01000001110011110110111 |
| 0xC2 | 648 | 01101000101100 | 0xF4 | 1 | 01000001110011110111000 |
| 0xE1 | 675 | 10010111100001 | 0xF5 | 1 | 01000001110011110111001 |
| 0x8D | 702 | 10010111100010 | 0xF6 | 1 | 01000001110011110111010 |
| 0x9C | 729 | 10010111100110 | 0xF7 | 1 | 01000001110011110111011 |
| 0xB1 | 729 | 10010111100111 | 0xF8 | 1 | 01000001110011110111100 |
| 0x81 | 756 | 10010111101110 | 0xF9 | 1 | 01000001110011110111101 |
| 0x7E | 837 | 11000001100111 | 0xFA | 1 | 01000001110011110111110 |
| | | | 0xFB | 1 | 01000001110011110111111 |

Codewords generated for 2GB data from dataset *DS-WikiEN* with default sampling rates.

Table B.2: Unused symbols in AHC compression.

| Hex | Char | Hex | Char | Hex | Char | Hex | Char | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|
| $0x00$ | NUL | $0x04$ | EOT | $0x05$ | ENQ | $0x06$ | ACK | $0x07$ | BEL |
| $0x08$ | BS | $0x0E$ | SO | $0x0F$ | SI | $0x11$ | DC1 | $0x12$ | DC2 |
| $0x13$ | DC3 | $0x14$ | DC4 | $0x15$ | ACK | $0x16$ | SYN | $0x17$ | ETB |
| $0x18$ | CAN | $0x19$ | EM | $0x1A$ | SUB | $0x1B$ | ESC | | |

# Appendix C

# MapReduce with RaPC Demonstrated

Listing C.1: MapReduce code segments for processing text in the original format.

```java
public class ProcessingClearText {
  public static class PCT_Mapper extends Mapper<Object, Text, Text,
      ↪ IntWritable> {
    private static Text outputKey = new Text();
    private static IntWritable outputValue = new IntWritable();
    public void map(Object key,Text value,Context context) throws Exception{
      String[] record = (value.toString()).split("\t");
      outputKey.set(record[1]);
      outputValue.set(Integer.parseInt(record[2]));
      context.write(outputKey, outputValue);
    }
  }

  public static class PCT_Reducer extends Reducer<Text, IntWritable, Text,
      ↪ IntWritable> {
    private IntWritable outputValue = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws Exception{
      int sum = 0;
      for (IntWritable value : values)
        sum += value.get();
      outputValue.set(sum);
      context.write(key, outputValue);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).
        ↪ getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: ProcessingClearText <input> <output>");
      System.exit(1);
    }
    Job job = Job.getInstance(conf, "MR with Clear Text");
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(ProcessingClearText.class);
    job.setMapperClass(PCT_Mapper.class);
    job.setReducerClass(PCT_Reducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    TextInputFormat.addInputPath(job, new Path(otherArgs[0]));
    TextOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Listing C.2: MapReduce code segments for processing RaPC-L2 compressed data.

```
public class ProcessL2CompressedText{
  public static class L2_Mapper extends Mapper<Object, Text, Text,
      ↪ IntWritable> {
    private static Text outputKey = new Text();
    private static IntWritable outputValue = new IntWritable();
    private static Iterable<String> records;
    public void map(Object key,Text value,Context context) throws Exception{
      records = Splitter.on('\n').split(value.toString());
      for (String record : records) {
        String fields[] = record.split("\t");
        outputKey.set(fields[1]);
        outputValue.set(Integer.parseInt(fields[2]));
        context.write(outputKey, outputValue);
      }
    }
  }

  public static class L2_Reducer extends Reducer<Text, IntWritable, Text,
      ↪ IntWritable> {
    private IntWritable outputValue = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context
        ↪ context) throws Exception {
      int sum = 0;
      for (IntWritable value : values)
        sum += value.get();
      outputValue.set(sum);
      context.write(key, outputValue);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).
        ↪ getRemainingArgs();
    if (otherArgs.length < 3) {
      System.err.println("Usage: ProcessL2CompressedText <record length> <
          ↪ input> <output>");
      System.exit(1);
    }
    RCaPCInputFormat.setRecordLength(conf, Integer.parseInt(otherArgs[0]));
    Job job = Job.getInstance(conf, "MR with RaPC-L2 Compressed Text");
    job.setInputFormatClass(RCaPCInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setJarByClass(ProcessL2CompressedText.class);
    job.setMapperClass(L2_Mapper.class);
    job.setReducerClass(L2_Reducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    RCaPCInputForma.addInputPath(job, new Path(otherArgs[1]));
    TextOutputFormat.setOutputPath(job, new Path(otherArgs[2]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Listing C.3: MapReduce code segments for processing RaPC-L2(L1) compressed data.

```
public class ProcessL2L1CompressedText {
  public static class L2L1_Mapper extends Mapper<Object, RaPCTextWritable,
      ↪ RaPCTextWritable, IntWritable> {
    private static RaPCTextWritable outputKey = new RaPCTextWritable();
    private static IntWritable outputValue = new IntWritable();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
      if (context.getCacheFiles().length > 0) {
        File modelFile = new File("./model.ip");
        T.buildModel(modelFile);
      }
    }

    public void map(Object key, RaPCTextWritable value, Context context)
        ↪ throws Exception{
      List<RaPCTextWritable> records = value.copyBytes().split('\n');
      for (RaPCTextWritable record : records) {
        List<RaPCTextWritable> fields = record.copyBytes().split('\t');
        outputKey.set(fields.get(0));
        outputValue.set(RaPC.T(fields.get(1)));
        context.write(outputKey, outputValue);
        }
      }
    }

  public static class L2L1_Reducer extends Reducer< RaPCTextWritable ,
      ↪ IntWritable, RaPCTextWritable , RaPCTextWritable > {
    private RaPCTextWritable outputValue = new RaPCTextWritable() ;
    public void reduce( RaPCTextWritable key, Iterable<IntWritable> values,
        ↪ Context context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values)
        sum += value.get();
      String temp = '\t' + String.valueOf(sum) + '\n';
      outputValue.set(temp.getBytes());
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).
        ↪ getRemainingArgs();
    if (otherArgs.length < 4 ) {
      System.err.println("Usage: ProcessL2L1CompressedText
          ↪ <record length> <model> <input> <output>");
      System.exit(1);
    }
    RCaPCInputFormat.setRecordLength(conf, Integer.parseInt(otherArgs[0]));
    Job job = Job.getInstance(conf, "MR with RaPC-L2(L1) Compressed Text");
    job.addCacheFile(new URI(otherArgs[1] + "#model.ip"));
    job.setInputFormatClass( RCaPCInputFormat .class);
    job.setOutputFormatClass( RCaPCSequenceFileAsBinaryOutputFormat .class);
    job.setJarByClass(ProcessL2L1CompressedText.class);
    job.setMapperClass(L2L1_Mapper.class);
    job.setReducerClass(L2L1_Reducer.class);
    job.setMapOutputKeyClass( RaPCTextWritable .class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass( RaPCTextWritable .class);
    job.setOutputValueClass( RaPCTextWritable .class);
    RCaPCInputFormat .addInputPath(job, new Path(otherArgs[ 2 ]));
    RCaPCSequenceFileAsBinaryOutputFormat .setOutputPath(job, new Path(otherArgs[
        ↪ 3 ]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

# Appendix D

# ASCII Table

Table D.1: Standard ASCII code table.

| Control | | Symbol | Number | Upper Case | | Lower Case | |
|---|---|---|---|---|---|---|---|
| NUL | DLE | SP | 0 | @ | P | ‘ | p |
| SOH | DC1 | ! | 1 | A | Q | a | q |
| STX | DC2 | " | 2 | B | R | b | r |
| ETX | DC3 | # | 3 | C | S | c | s |
| EOT | DC4 | $ | 4 | D | T | d | t |
| ENQ | NAK | % | 5 | E | U | e | u |
| ACK | SYN | & | 6 | F | V | f | v |
| BEL | ETB | ’ | 7 | G | W | g | w |
| BS | CAN | ( | 8 | H | X | h | x |
| HT | EM | ) | 9 | I | Y | i | y |
| LF | SUB | * | : | J | Z | j | z |
| VT | ESC | + | ; | K | [ | k | { |
| FF | FS | , | < | L | Space | l | \| |
| CR | GS | – | = | M | ] | m | } |
| SO | RS | . | > | N | ^ | n | ~ |
| SI | US | / | ? | O | _ | o | DEL |

Appendix E

# Efficient Cloud Server Consolidation Supported by Online Data Analysis Services

The deployment of big data analytics platforms such as Hadoop requires a cluster of servers. Depending on the characteristics of the analysis, the resource requirements on server configuration often vary greatly. As cloud computing becomes widely available, it has become the main enabling technology for big data analysis due to its dynamic provisioning and cost-efficiency. Many existing private and commercial big data solutions are already deployed on clouds, such as IBM SmartCloud [IBMb], OpenStack Sahara [Sta], Amazon Elastic MapReduce (EMR) [Ama] and Cloudera [Clo]. Moving optimization from source data to computing nodes. We also look at the efficient use of cloud resources in this work.

As the utility computing paradigm requires massive server deployment, one of the main concerns for a cloud service provider is the operational cost, especially the cost of power consumption. Survey research indicates that servers in many organizations typically run at less than 30% of their full capacity [BH07] [SM10]. Thus, it is possible to reduce power consumption of the hardware by means of allocating more Virtual Machines (VMs) to less hosts. This is known as server consolidation. It is one of the key techniques used for energy saving in clouds. The basic principle of server consolidation is to allocate as many VMs on a physical server as possible, while satisfying various constraints specified as part of the system requirements. Depending on business strategy or user preference, previous work has focused on improving VM performance and availability [JPE+11], scalability [JLH+12] [BCF+12] [MPZ10], energy conservation [BAB12] [VAN08], Service Level Agreement (SLA) violation prevention [BB12], VM live migration cost [CFH+05] [LXJ+11] [VBVB09] or some combination of these [GGP12] [XF10].

Additionally in the commercial use of cloud computing, SLA is one of the main methods to deal legally with Quality of Service (QoS) guarantees such as service availability and performance and, as such, has attracted the attention of both consumers and service providers. Technically, QoS can be guaranteed through resource provisioning through pre-defined policies and/or dynamic methods such as resource utilization predictions. Due to the dynamic and heterogeneous nature of cloud services, a dynamic method is preferable. However, predictions are often inaccurate. An improved prediction accuracy is achieved in this work by allowing each individual VM to build its own utilization model on demand over an appropriate time horizon. Thus, the utilization model can reflect accurately the characteristics of resource usage of the VM. The utilization model is supported by a sophisticated data analysis system implemented as a service. More specifically, the *R* open source data analysis framework [The] is employed as a modeling and decision support system. The *R* Decision Support System (rDSS) is designed and deployed as a cloud-based solution. It provides services to the server consolidation algorithms. In this work [DH13b], both energy conservation and SLA violation prevention are considered.

We employ several advanced data modeling and forecasting techniques. Results from simulation-based experiments show that the proposed server consolidation algorithm, supported by the data analysis service, can effectively reduce power consumption, the number of VM migrations and SLA violation. It also compares favourably with other start-of-the-art heuristic algorithms.

## E.1 System Architecture

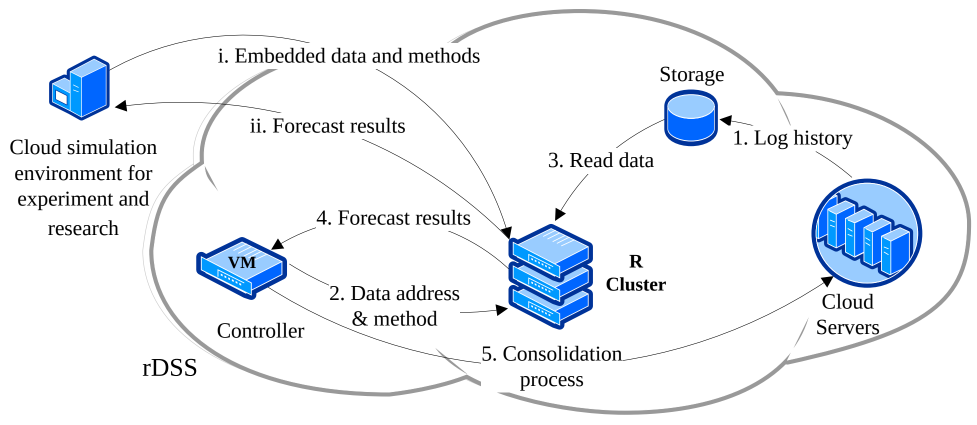


Figure E.1: The proposed rDSS system architecture.

The rDSS system is a cloud-based solution. Resource utilization information from each VM and host is logged to a centralized location as shown in Figure E.1. The data collection process can be carried out by a Hypervisor or a third party software. Data analysis services that employ the *R* framework as an engine are pre-packaged Virtual Machines which can be easily deployed on clouds. The number of data analysis servers can be scaled out/in on demand depending on the number of VMs and Servers need to be monitored. In the rDSS, servers are divided into groups. Each group is managed by a *Controller* component. The *Controller* has three responsibilities. Firstly, it segments the VM/Server utilization data in a specified length. This segmented data is then used for building resource forecasting models. Secondly, the *Controller* forwards the address of the data stored in the cloud and a specified modeling algorithm to the rDSS. Information is sent programmatically by calling program functions which are embedded in the *R* language clauses. Communication can also be done asynchronously via a queuing system. In this case, information will be sent and received as messages through queues. The asynchronous communication approach is particularly useful when the dataset is large. The returned results from the rDSS can also be received synchronously. Finally, based on the forecast model and prediction results returned from rDSS, the *Controller* carries out the VM consolidation process.

## E.2 Problem Formulation

Server consolidation falls into the field of multi-objective optimization. It is often formulated as a Bin Packing or a Constraint Programming problem. [CZS$^{+}$11] proposed an Effective Sizing guided server consolidation algorithm. The consolidation process is modeled as a bin packing problem. The proposed Effective Sizings are calculated by computing least workload correlations with other VMs on the target Server. It is based on a strong assumption that workload distributions are stationary over time. [JPE$^{+}$11] is an another algorithm that the complexity of the optimization is reduced by grouping VMs into location-aware clusters. As server consolidation is often an NP-complete task, many researchers employ heuristic algorithms in order to provide sub-optimal solutions in a timely fashion. A Power Aware Best Fit Decreasing (PABFD) heuristic was proposed in [BAB12]. The PABFD algorithm starts by identifying over/under utilized Servers, then selects VMs from an over/under utilized Server to migrate to other Server in order to reduce the number of bins (active Servers) used. Each selected VM is then pseudo-assigned to a Server. Power consumption for each trial is calculated by the given power consumption models of Servers and the CPU requirements of the VM. This process iterates through all active Servers, then a target Server with minimum power consumption is selected. All selected VMs go through the same process sequentially until they are all assigned to a Server.

More advanced heuristic algorithms such as Genetic Algorithms (GAs) are also used [XF10]. GAs are known to be effective for optimization problems with large search domains. The authors [XF10] consider power consumption and cooling cost as the main factors in their multi-objective optimization formulation. They formulate Servers and VMs as chromosomes in the GA context, then go through the conventional process of crossover, mutation and fitness in order to find an optimal solution. The final solution is also a sequence of numbers (chromosomes) that represents a new mapping of VMs to Servers. Although GAs may provide better solutions than simple heuristic algorithms (such as First Fit Decreasing and Best Fit Decreasing), but it is not able to provide optimal solutions in a timely fashion. Another weakness of GAs, pointed out by the authors, is that the GA approach has high possibilities of causing a large amount of unnecessary VM migrations in cloud.

In parallel, recent research also found that network communication consumes a considerable portion of energy in cloud data centres [MPZ10] [BCF+12]. This is not surprising if one considers moving a number of VMs with memory footprint ranging from several hundred megabytes to tens of gigabytes. Using VM migration as a technique for VM consolidation can therefore cause both network performance and VM performance to degrade considerably [LXJ+11][CFH+05][VBVB09] consequently affects QoS. [MPZ10] proposed a traffic-aware algorithm for server consolidation. The authors formulate the server consolidation process as a Quadratic Assignment Problem and makes their solution aware of network topologies and network traffic patterns. In order to make the algorithm feasible for real deployment, VMs are grouped into clusters; further simplifications are also made for homogeneous VM configuration; and each server is only able to contain the same number of VMs. Simple statistical and polynomial curve fitting [BB12] and more advanced technique using *Kalman* filters [KKH+08] are also used for server consolidation. [KKH+08] uses a trained *Kalman* filter to produce estimates of the number of workload requested and to forecast the future state of the system. The accuracy of the algorithm relies on a properly trained filter. It is not clear how the filter can be *properly* trained and to what extent the filter is *properly* trained.

$$\begin{aligned}
\min \quad & \sum_{i\in V}\sum_{j\in H} p_{ij}v_{ij} + \sum_{j\in H} f_j h_j \\
\text{s.t.} \quad & \sum_{i\in V} r_i v_{ij} \leqslant Ch_j \quad && \forall j \in H \\
& \sum_{i\in V} \{\hat{r}_{i(t+n)}\} v_{ij} \leqslant Ch_j \quad && \forall j \in H \\
& \sum_{j\in H} v_{ij} = 1 \quad && \forall i \in V \\
& v_{ij} \leqslant h_j \quad && \forall i \in V, j \in H \\
& v_{ij} \in \{0,1\} \quad && \forall i \in V, j \in H \\
& h_j \in \{0,1\} \quad && \forall j \in H
\end{aligned}$$

In contrast, the objectives of rDSS are to minimize both power consumption and SLA violation. Given a set $H$ of Servers, a set $V$ of virtual machines in a cloud data centre and power consumption models for each Server, the objectives are to decide how to rearrange $V$ on $H$ such that the total power consumption in the data centre is minimized, while the SLA violation rate is kept as low as possible. All $v \in V$ requirements $r_i$, such as CPU, memory and storage, must be satisfied by the target Server, all $v \in V$ predicted CPU requirements $\{\hat{r}_{i(t+n)}\}$ must be satisfied by the target Server in order to minimize SLA violations. Each $h$ has a resource capacity limit $C$. The total power consumption is the sum of power $p_{ij}$ consumed by CPUs of each VM $i$ on Server $j$, plus a fixed power $f_j$ consumed by the other components of Server $j$, such as memory and I/O, etc. Let $h_j = 1$ represents choosing Server $j$ to be switched on and $0$ otherwise. Also, let $v_{ij} = 1$ represents the assignment of VM $i$ to Server $j$ and $0$ otherwise. The mathematical model is outlined below. The first constraint enforces the capacity limit on each Server. The second constraint minimizes SLA violations. The third constraint ensures that each VM is assigned to exactly one Server. The fourth constraint guarantees a Server to be switched on if and only if there is a VM assigned to that Server. The last two constraints indicate that the state of a VM or Server is either on or off (Equation E.1).

### E.2.1 SLA Violation Minimization

SLA violation has various aspects. One of the main causes of SLA violation is that the requested resources from VMs can not be satisfied by the resource providers (Servers). SLA violation minimization is accomplished in two parts.

In the first part, forecast models are built for each VM based on a certain length of historical data. The forecast model is then used to predict the future CPU requirements for each VM. The SLA is controlled as follows. Let the matrix $A \in \mathbb{R}^{+m*n}$ denotes the total $m$ VMs in a cloud. Each VM associates with a list which contains $n$ step-ahead forecast values. The number of steps is adjusted according to the consolidation process frequency, where $\hat{r}_{i(t+n)}$ denotes the predicted CPU requirements for VM $i$ at time $t+n$. For all VMs that have been placed or are going to be placed on Server $j$ construct a matrix $A'$, $A'_j \subseteq A$, $i \leqslant m$:

$$A'_j = \begin{bmatrix} \hat{r}_{0(t+1)} & \hat{r}_{0(t+2)} & \cdots & \hat{r}_{0(t+n)} \\ \hat{r}_{1(t+1)} & \hat{r}_{1(t+2)} & \cdots & \hat{r}_{1(t+n)} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{r}_{i(t+1)} & \hat{r}_{i(t+2)} & \cdots & \hat{r}_{i(t+n)} \end{bmatrix}$$

Such that,

$$SLA(A'_j) \models \{\alpha \sum_{k=0}^{i} \hat{r}_{k(t+n)} \leqslant Ch_j, \quad \forall n\} \tag{E.1}$$

Forecasting often has limited accuracy. The second part of SLA protection is to reserve a certain amount of resources on each Server to tolerate forecast errors and accommodate sudden bursts in CPU requests as indicated by a factor $\alpha$ in Equation E.1. The resource reservation strategy is outlined as follows. If the available resource on a Server is less than the reserved resources (resource buffer), the Server will be seen as over utilized, then one or more VMs will be selected to migrate to other Server(s). The results of VM(s) migration to the target Server(s) must not violate the condition specified in Equation E.1. However, VM migration often results in VM performance degradation, extra load on the network and more energy consumption [LXJ+11] [CFH+05] [VBVB09]. Additionally, [BB12] states that migrating smaller VMs is desirable. [LXJ+11] asserts that the time required for VM live migration is mainly determined by memory size, memory dirtying rate and network bandwidth. Based on these conclusions, the principle of smallest memory size first is used in the selection of VMs for migration. For simplicity, it is assumed that the memory dirtying rate and network bandwidth are kept constant. In this work, we reserve resources on each Server using a fixed value.

As the proposed VM placement algorithm relies heavily on the forecast results, an accurate forecast model is at its foundation. Since VMs are continuously running in a cloud, the sampled CPU utilization of each VM can be treated as a time series data. It should be noted that due to the heterogeneity of workload, the time series of VMs often exhibit different properties, hence we need an adaptive way of building forecast models without prior knowledge about the types of workload. Employing the powerful *R* framework as the decision support system allows us to produce forecast results based on five different models including Auto Regressive Integrated Moving Average model (ARIMA), Exponential sTate Space model (ETS), Independent Identically Distributed model (IID), Random Walk model (RW) and Structural Time Series model (STS).

A prerequisite for the proposed algorithm is to establish connections between VMs and rDSS services. Once the connections are established, VMs remain connected during their lifetime and the algorithm maintains a map of VMs to connections. The map is used as an input to the optimization algorithm. The events of an VM joining or leaving will correspond to a map refresh action. The VM placement process follows the same principle as the Power-aware Best Fit Decreasing (PABFD) algorithm [BB12]. The differences are that any successful placement needs to satisfy both hardware requirements (such as memory and storage) and SLA requirements. An over utilized Servers are determined by examining the condition $SLA(A'_{host})$. Given the power consumption model of each Server, the reason for choosing PABFD is that it allows VMs to be placed on more power efficient Servers in a heterogeneous environment. Given a list of Servers $H$, a list of VMs $V$ and Server power consumption model $PM$, the process of placing VMs selected from over utilized Servers is outlined in Algorithm 3. The process for determining under utilized Servers is described in [BB12].

**Algorithm 3:** Server consolidation algorithm

```
Input: $H, V, M, PM$
$A \longleftarrow construct(M, V)$
for $host \in H$ do
    $A' \longleftarrow construct(host, A)$
    if $overUtilized(A')$ then
        $vm \longleftarrow selectVM(host)$
        $E \longleftarrow \infty$
        $candidateHost \longleftarrow \varnothing$
        foreach $activeHost \in H$ do
            $A' \longleftarrow construct(activeHost, vm, A)$
            if $SLA(A')$ & $reqHW(activeHost, vm)$ then
                $E' \longleftarrow energy(activeHost, vm, PM)$
                if $E > E'$ then
                    $E = E'$
                    $candidateHost = activeHost$
        if $candidateHost = \varnothing$ then
            $candidateHost \longleftarrow selectIdelHost(H)$
        $deploy(candidateHost, vm)$
```

## E.3 Evaluation

Several sets of simulations are used for conducting the evaluation on the effectiveness of the proposed server consolidation algorithm and the rDSS system. The simulated environment is Infrastructure-as-a-Service (IaaS). It consists of 80 HP ProLiant ML110 G4 and G5 Servers, randomly selected to build a heterogeneous cloud environment. Power consumption models of the Servers are shown in Figure E.2. Depending on the experiment, the number of VMs ranged from 25 to 150 with memory assignment uniformly distributed in the range [256MB, 1GB]. A set of mixed, real-world server workloads [BB12] is used in the experiments. The type of servers include Domain Name Servers (DNS), Dynamic Host Configuration Protocol servers (DHCP), File Transfer Protocol servers (FTP), Hypertext Transfer Protocol servers (HTTP), Proxy servers, Database servers and Application servers. Each server workload is given to a dedicated VM during the simulation. The evaluation is twofold. The proposed consolidation algorithm with five different prediction models is firstly evaluated against the PABFD algorithm, then the characteristics and performance of the rDSS system are evaluated.

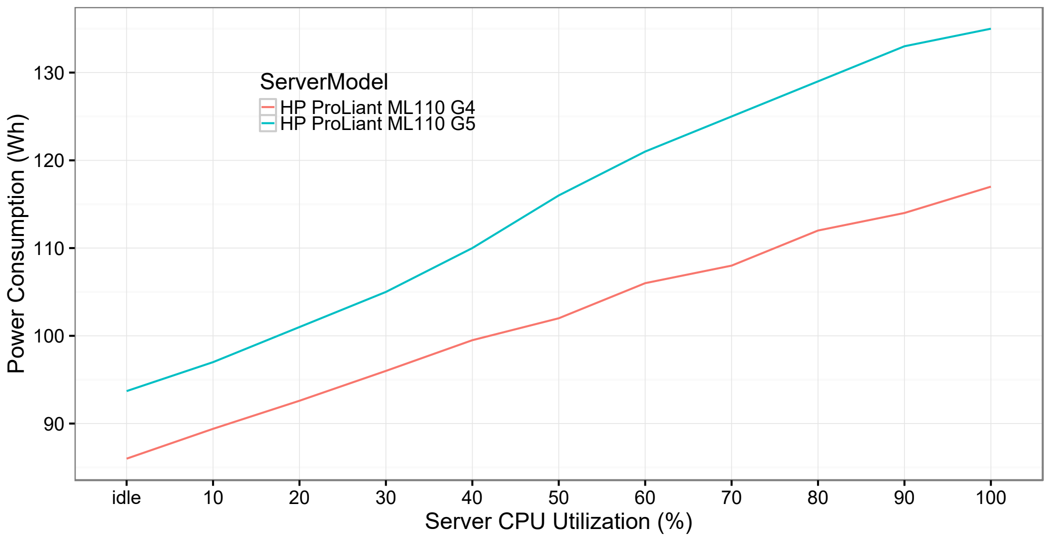


Figure E.2: Server power-models used in the experiments.

## E.3.1 Algorithm Performance

We use three sets of experiments to evaluate our One-step ahead Forecast-based Power-aware Best Fit Decreasing (F1PaBFD) algorithm. All of the experiments are configured with one hundred VMs randomly deployed on eighty Servers, initially. The experiments simulate a cloud environment continuously operational for six hours, with the consolidation frequency set to five minutes. We use the default fitted models generated by the corresponding algorithms in the *R* framework.

The new F1PaBFD algorithm is evaluated using the five forecast models, hence named F1PaBFD-ARTIMA, F1PaBFD-ETS, F1PaBFD-IID, F1PaBFD-STS and F1PaBFD-RW, respectively. These five versions of our algorithms are compared with the standard state-of-the-art PABFD heuristic algorithm which is based on the Local Regression and Minimum Migration Time techniques [BB12]. Forecast models are built for each individual VM based on two hours of historical data. Figure E.3(a)(c)(e) and (g) show the results from the first set of experiment. A set of mixed, real-world server workloads is given to each VM during the simulation. The workload is directly mapped to the CPU utilization of each VM. Overall, the average CPU utilization of all Servers is approximately 10.74%. Figure E.3(a)(c)(e) and (g) show the total power consumption, number of VM migrations, SLA violation per active Server and overall SLA violation for each of the F1PaBFD based algorithms in comparison with the existing algorithm. We observe that our F1PABFD based algorithm is able to save 0.03 $\sim$ 0.17 electricity units (KWh) over the six hours of operation comparing to the PABFD. It is also shown that F1PaBFD can reduce the total number of VM migrations significantly for all algorithms apart from the algorithm that uses Random Walk. Among the algorithms, F1PaBFD-IID produces the lowest power consumption and minimum number of VM migrations (39%

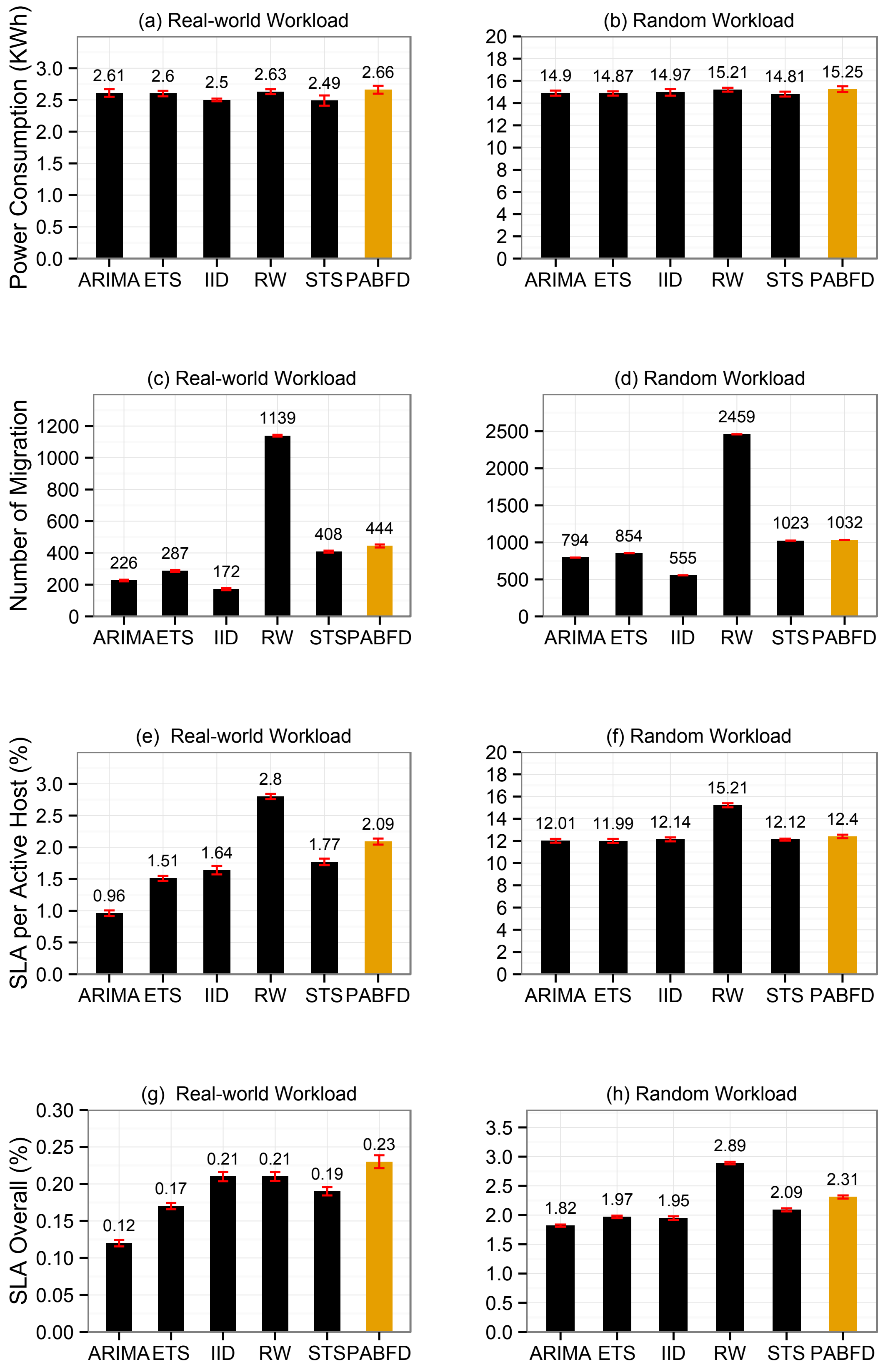


Figure E.3: Experimental results from real-world server-workload and randomly generated workload.

less than PABFD). However, this gain is offset by higher SLA violation as shown in Figure E.3(e) and (g). In contrast, based on sophisticated forecasting techniques, we can effectively prevent the SLA violation by reducing SLA violation per active Server to the best result of 0.96% (F1PaBFD-ARIMA) and overall SLA violation to the best result of 0.12%, comparing to 2.09% and 0.23% given by the PABFD, respectively. This leads to a choice of statically or dynamically selecting among the F1PaBFD algorithms with regard to the deployment strategies.

In the second experiments, we evaluate the robustness of our algorithms. We perform exactly the same experiments but with random workload. The artificially generated random workload reflects approximately 53.2% CPU utilization on average. In Figure E.3(b), the power consumption patterns are similar to that produced in Figure E.3(a). Because the random workload is heavier, thus the power consumption is higher. With the random and heavier workload, our algorithms start saving more energy, $0.03 \sim 0.44$KWh. In Figure E.3(d), we still observe a significant drop in the number of VM migrations. In contrast with the real-world workload, using the random workload greatly increases SLA violations as shown in Figure E.3(f) and (h). The increase in SLA violations is mainly due to the randomness of the workload that affects the accuracy of the forecasting results.

In the third experiment, we eliminate the F1PaBFD-RW due to having the highest number of VM migrations and F1PaBFD-ETS due to having the lowest performance as shown in Figure E.5. We evaluate how the size of workload affects our algorithms. We generate eight sets of workloads artificially based on the real-world server workloads used in the first experiment. To preserve the characteristics of the original workloads as much as possible, such as trends and periodicity, etc., we add constant values to each workload, then gradually increase the weight of the average utilization of the workload from 10.74% (original) up to 90%. In Figure E.4, we observe that all F1PaBFD based algorithms consume less power (ranging from $0.03 \sim 1.14$KWh) than the non-forecast based PABFD (as shown in Figure E.4(*Default*)). A significant power drop is observed at an average workload at 90%. This is because when the average workload (average CPU utilization of each VM) is reaching the full capacity of a Server, the changes in the CPU utilization curves become more smooth. Our forecasting then becomes more accurate. Additionally, when the level of the average workload is high, the bursty level of CPU utilization becomes smaller which results in lower SLA violation as shown in Figure E.4. In contrast, the SLA violation becomes higher when average workload becomes lower. This is because when CPU utilization of each VM is lower, more VMs will be assigned to a Server, thus it increases uncertainty of resource requirements. An improvement can be made by dynamically reserving more resources on each Server according to the level of the average workload. This is planned for future implementation. Overall, our F1PaBFD algorithms perform generally better than the original PABFD, especially in reducing the number of VM migrations. In the middle facet of Figure E.4, we observe that

F1PaBFD-IID, -STS and -ARIMA algorithms can reduce the number of VM migrations by $\{3, 1.5, 1.5\}$ times on average, comparing to PABFD. Among the algorithms, F1PaBFD-STS and F1PaBFD-ARIMA perform consistently well on power conservation, reducing the number of VM migrations and preventing SLA violation. F1PaBFD-IID shows better ability to reduce the number of VM migrations. Considering that F1PaBFD-ARIMA algorithm is much slower than F1PaBFD-STS and F1PaBFD-IID as shown in Figure E.5, F1PaBFD-STS and F1PaBFD-IID are recommended. Depending on the conditions of the network in a cloud environment, the F1PaBFD-IID may be preferred over F1PaBFD-STS due to less VM migrations. Thus, there is an opportunity for research in dynamic algorithm switching based on network conditions.

### E.3.2 RDSS System Performance

In the experimental environment, two *R* servers (VMs) are deployed as the decision making engine. Each of the *R* servers is configured with 1GB memory, single core 2.2GHz processor, 60GB local hard disk. The *R* servers are connected to the simulation environment over FastEthernet (100Mb/s). VMs created in the simulation environment are randomly assigned to rDSS servers initially. The *R* packages are compiled based on Linux kernel *version 3.20.0*, 64-bit architecture. We choose to use *R version 2.15.1*. We also configure *Rserve* services (*version 1.7.0*) on each VM so that our simulation environment can talk to the decision making servers using Java. We use synchronous communication through *Rserve* for all experiments. Historical data of VMs and *R* commands are encapsulated into Java objects and they are sent to the decision making servers as serialized objects over a Transmission Control Protocol/Internet Protocol (TCP/IP) socket.

Figure E.5 shows the algorithm performance on a single rDSS server including F1PaBFD-ARIMA, -ETS -IID, -RW and -STS. Using the same length of historical data, forecasting based on the ETS and ARIMA model are relatively time consuming. On the other hand, forecasting based on the IID, Random Walk and StructTS models are extremely fast. The best results are obtained from the F1PaBFD-IID algorithm for forecasting 150 VMs in 0.75 seconds. Results from Figure E.5 can also be used for determining the number of rDSS servers needed in order to meet the desired performance. In this experiment, we gradually increase the number of VMs from 25 to 150. Overall, we observe a linear performance degradation with an increasing number of VMs. This is because the forecasting models in the *R* baseline version are implemented as a single threaded program. Every request is dealt with sequentially. This is also the main reason we configure virtual machines with a single processor for each rDSS server. There are exceptions for some packages which support parallel processing such as the automatic ARIMA process. Figure E.6 shows the system performance using multi-core processors for the automatic ARIMA process. We observe that performance increases slowly when

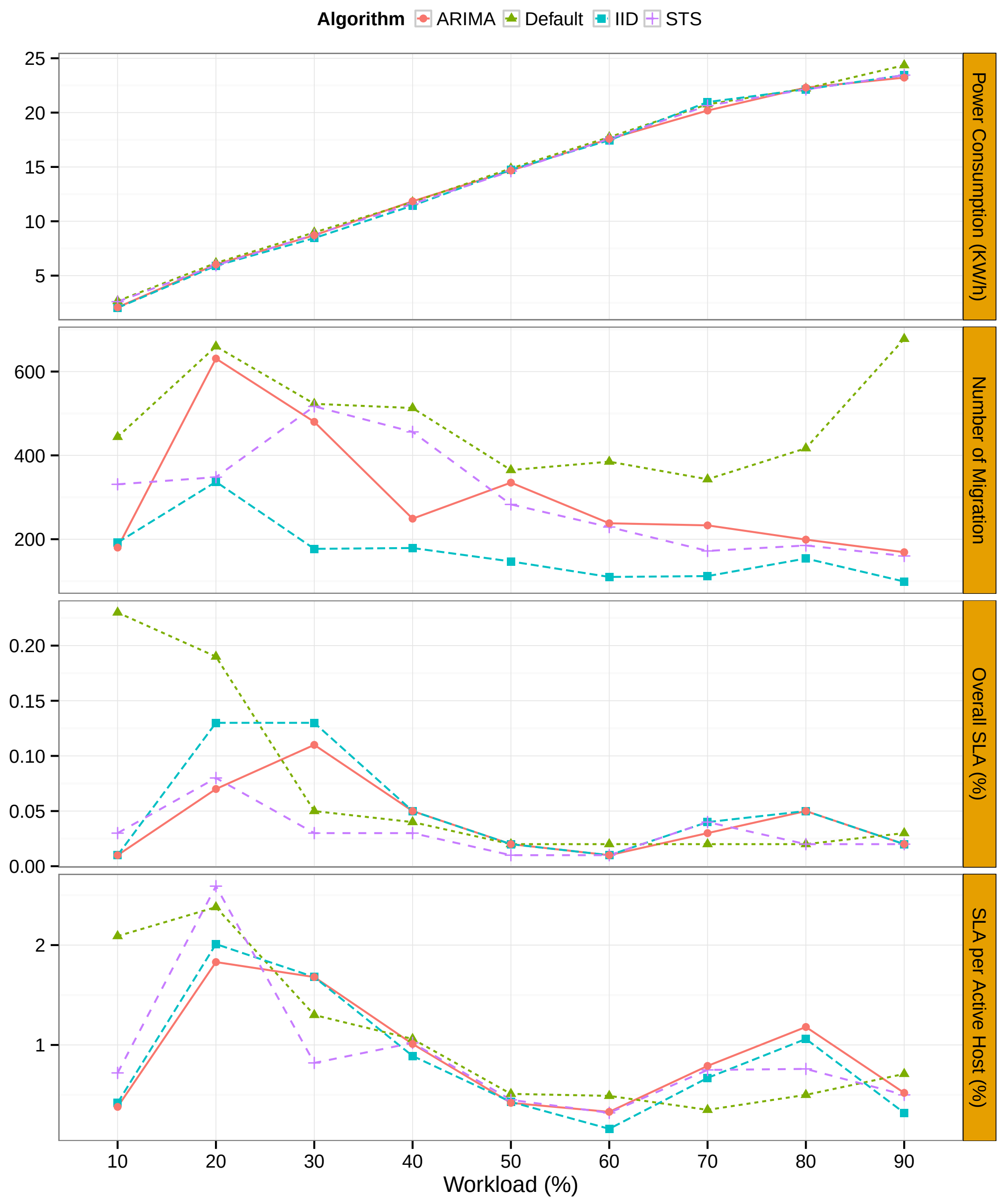


Figure E.4: Evaluation of the proposed F1PaBFD algorithm.

the number of cores are greater than four. This is because the Hyper-Threading technology is seen as physical processors by the underlining operating system, but it does not preform as well as physical processors. In future work, other high performance solutions will be investigated, for example, the *Revolution R* [Nie11].

The average run time for each algorithm with the historical data length of $\{2, 4, 8\}$ hours is shown in Figure E.7. In Figure E.7, the performance of F1PaBFD-IID, -RW and -STS algorithms exhibit no clear changes when data length increases. On the other

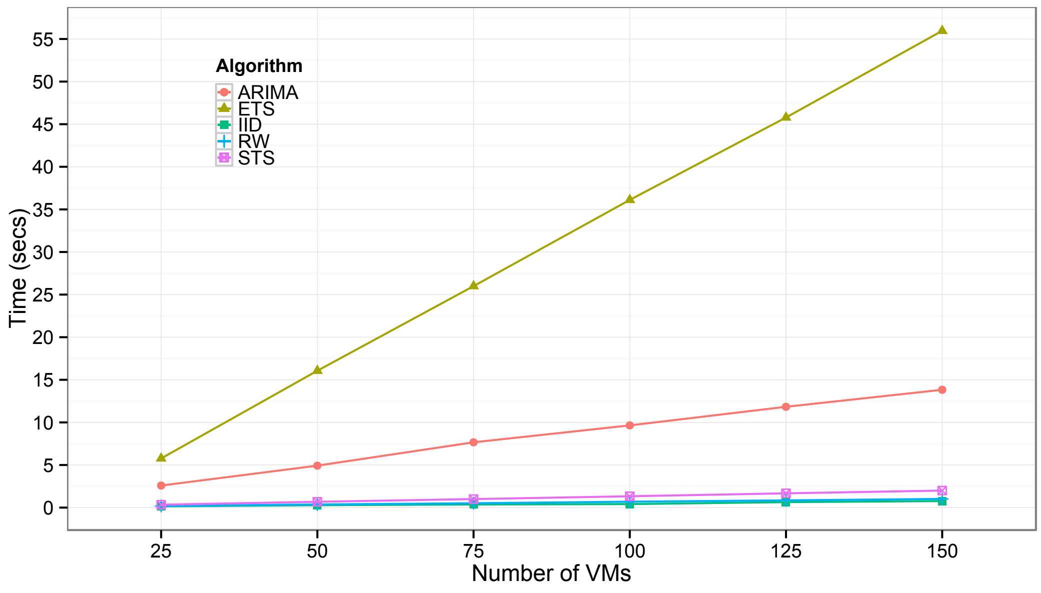


Figure E.5: Evaluation of F1PaBFD performance .

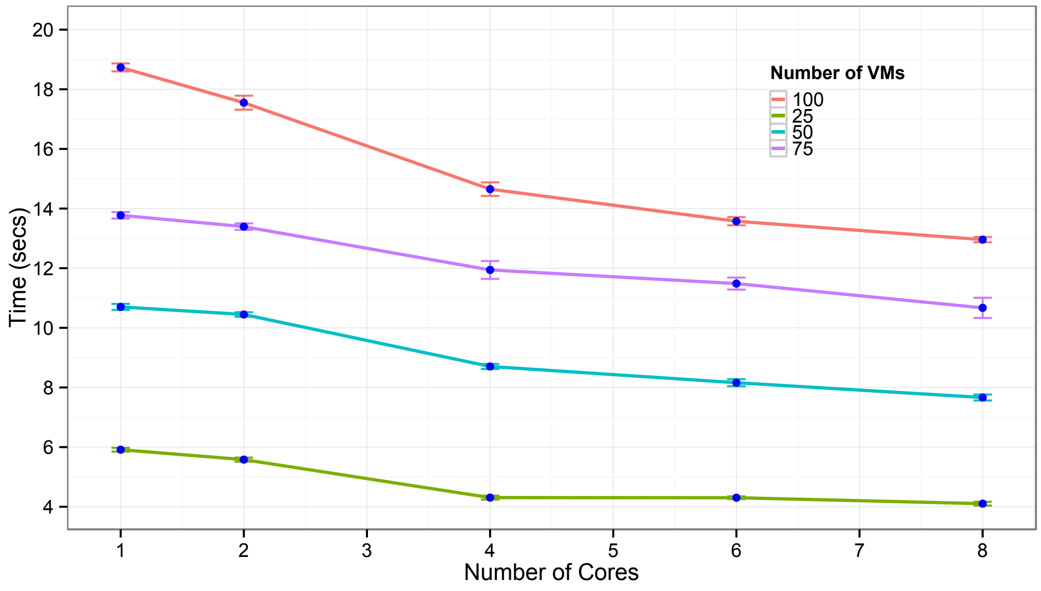


Figure E.6: Evaluation of performance on single node rDSS system with multi-core processors using automatic ARIMA algorithm.

hand, F1PaBFD-ARIMA and -ETS performance decrease rapidly. The results depend solely on the complexity of the algorithms. Figure E.8 shows the memory footprint size increases slightly when the number of VMs increases. The results are heavily influenced by the experimental methods. In the experiment, at the beginning of the simulation, we create connections for each VM to the rDSS system. Each connection remains connected until the simulation ends. Each connection consumes approximately 140KB memory. We also do memory garbage collection at the end of each request,

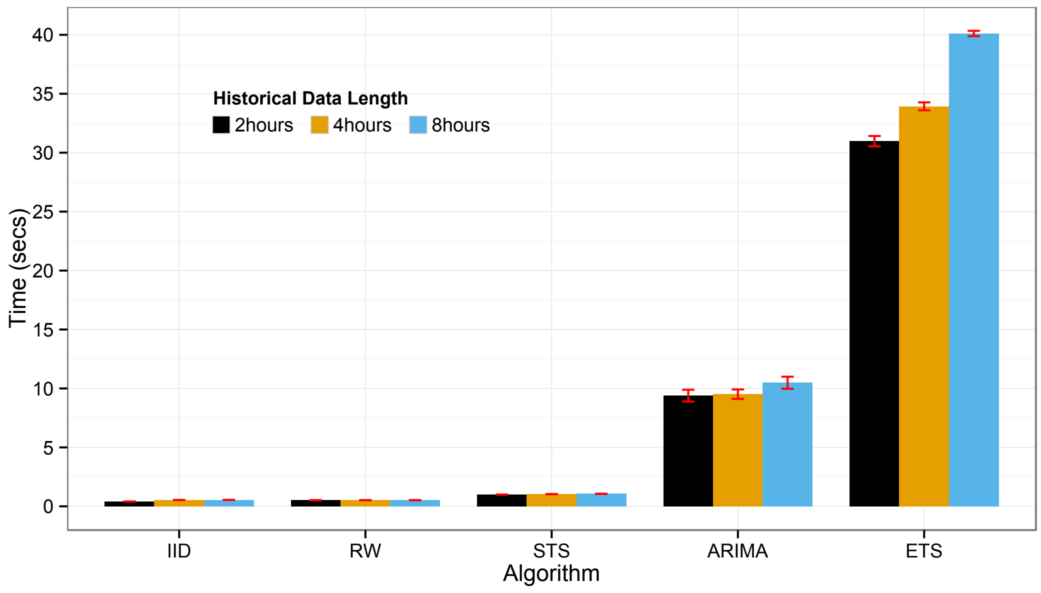


Figure E.7: Evaluation of performance on single node rDSS system.

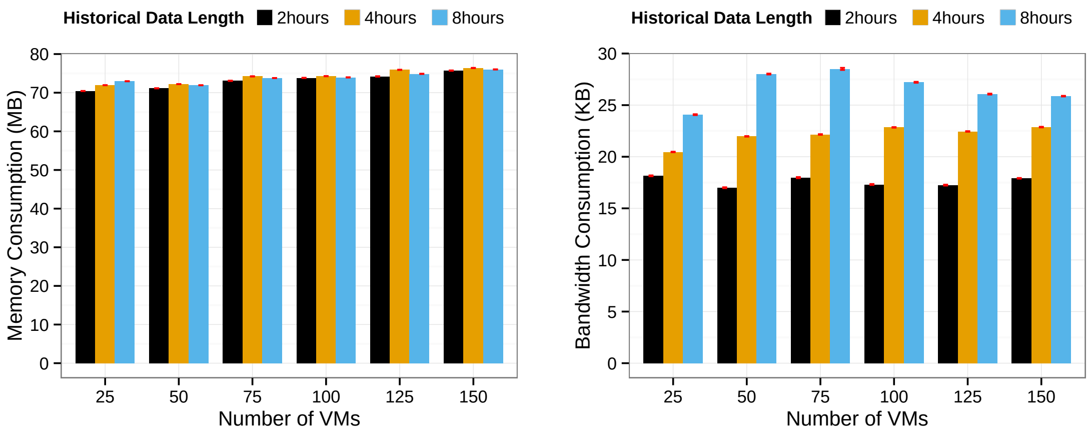


Figure E.8: Memory and bandwidth requirements for single node rDSS system.

so that the system memory does not get dirty after executing for a while. Memory requirement can be estimated by $140*c+D+P$ in KB, where $c$ is number of connections, $D$ is data length and $P$ is the memory required by the $R$ framework itself. Network bandwidth consumption depends solely on the length of the historical data used for building the models.

## E.4 Conclusion

In this work, we introduced a forecast-based power-aware server consolidation algorithm. The main objectives are to maximize energy efficiency while keeping SLA violation as low as possible. The rDSS system is provided as a service and deployed on clouds. It can be used as a plug-in service or integrated into existing clouds. The rDSS builds forecast models for each individual VM and predictions are made as to their future CPU resource requirements. We evaluated the new forecast based algorithms using several advanced modeling techniques including auto ARIMA, ETS, STS, IID and RW. The simulation based evaluations were conducted for the proposed versions of the algorithm with respect to their consistency, robustness and performance. The experiments also included, for comparison, a state-of-the-art alternative VM placement algorithm. The results show that the proposed approach can significantly reduce power consumption, the number of VM migrations and SLA violation comparing to other heuristic algorithms. We also evaluated the rDSS system in terms of its performance.

# References

[ABKY15] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. Transparent data deduplication in the cloud. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 886–900, New York, NY, USA, 2015. ACM.

[AK93] Tetsuo Asano and Naoki Katoh. Number theory helps line detection in digital images an extended abstract. In *Algorithms and Computation*, pages 313–322. Springer, 1993.

[AL98] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference, 1998. DCC '98. Proceedings*, pages 119–128, March 1998.

[AL13] Nima Asadi and Jimmy Lin. Fast candidate generation for real-time tweet search with bloom filter chains. *ACM Trans. Inf. Syst.*, 31(3):13:1–13:36, July 2013.

[AM97] Z. Arnavut and S.S. Magliveras. Block sorting and compression. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 181–190, March 1997.

[Ama] Amazon EMR. `https://aws.amazon.com/elasticmapreduce/`. [Accessed on 12-August-2015].

[Ama13] Xavier Amatriain. Big & personal: Data and models behind netflix recommendations. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '13, pages 1–6, New York, NY, USA, 2013. ACM.

[Apaa] Apache Avro. `http://avro.apache.org/`. [Accessed on 20-August-2015].

[Apab] Apache Cassandra. `http://cassandra.apache.org/`. [Accessed on 20-September-2015].

[Apac] Apache Commons CSV . http://commons.apache.org/proper/commons-csv/. [Accessed on 20-September-2015].

[Apad] Apache Hadoop. https://hadoop.apache.org/. [Accessed on 20-September-2015].

[Apae] Apache HBase. http://hbase.apache.org/. [Accessed on 29-August-2015].

[Apaf] Apache Mahout. http://mahout.apache.org/. [Accessed on 02-October-2015].

[Apag] Apache Storm. http://storm.apache.org/. [Accessed on 26-August-2015].

[Apah] Apache Tez. http://tez.apache.org/. [Accessed on 26-August-2015].

[Apai] Apache Xerces. http://xerces.apache.org/. [Accessed on 21-September-2015].

[Apaj] Apache ZooKeeper. https://zookeeper.apache.org/. [Accessed on 26-August-2015].

[Apa15] Apache Spark. http://spark.apache.org/, 2009 - 2015. [Accessed on 11-August-2015].

[AtS15] 2015 hadoop maturity survey results. Report, AtScale, 2015.

[Baa96] R. Harald Baayen. The effects of lexical specialization on the growth curve of the vocabulary. *Comput. Linguist.*, 22(4):455–480, December 1996.

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28:755 – 768, 2012.

[BB01] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 26–33, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics.

[BB12] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012.

[BbXb15] Miao Bei-bei and Jin Xue-bo. Compressing sampling for time series big data. In *Control Conference (CCC), 2015 34th Chinese*, pages 4957–4961, July 2015.

[BCC$^+$00] Adam L. Buchsbaum, Donald F. Caldwell, Kenneth W. Church, Glenn S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: An experimental approach. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 175–184, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[BCF$^+$12] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera. A stable network-aware vm placement for cloud systems. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 498–506, May 2012.

[BDM13] Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors. *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*. Springer, 2013.

[BFG03] Adam L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. Improving table compression with combinatorial optimization. *J. ACM*, 50(6):825–851, November 2003.

[BFG09] Philip Bille, Rolf Fagerberg, and Inge Li Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Trans. Algorithms*, 6(1):3:1–3:14, December 2009.

[BH07] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, August 2010.

[bJfLSW13] Xue bo Jin, Xiao feng Lian, Yan Shi, and Li Wang. Data driven modeling under irregular sampling. In *Control Conference (CCC), 2013 32nd Chinese*, pages 4731–4734, July 2013.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BLR$^+$11] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings. In *Proceedings of the Twenty-second Annual ACM-*

*SIAM Symposium on Discrete Algorithms*, SODA '11, pages 373–389. SIAM, 2011.

[BO14] Chongke Bi and K. Ono. 2-3-4 combination for parallel compression on the k computer. In *Visualization Symposium (PacificVis), 2014 IEEE Pacific*, pages 281–285, March 2014.

[BOY14] Chongke Bi, K. Ono, and Lu Yang. Parallel pod compression of time-varying big datasets using m-swap on the k computer. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 438–445, June 2014.

[BSTW86] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, April 1986.

[BW94] M. Burrows and D.J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, May 1994.

[Bzi] Bzip Compressor. `http://www.bzip.org/`. [Accessed on 12-January-2015].

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[CCS12] Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. Business intelligence and analytics: From big data to big impact. *MIS Q.*, 36(4):1165–1188, December 2012.

[CDG$^+$08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[CFH$^+$05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286. USENIX Association, 2005.

[CH87] Gordon Cormack and Nigel Horspool. Data compression using dynamic markov modelling. *Computer Journal*, 30(6):541–550, October 1987.

[Cha00] Anindya Chatterjee. An introduction to the proper orthogonal decomposition. *Current Science*, 78(7):808–817, April 2000.

[Clo] Cloudera. `http://www.cloudera.com`. [Accessed on 26-August-2015].

[Col14] E. Collins. Intersection of the cloud and big data. *Cloud Computing, IEEE*, 1(1):84–85, May 2014.

[Con09] Genome Reference Consortium. Grch37 genome reference consortium human reference 37 (gca 000001405.1). `https://hgdownload.cse.ucsc.edu/goldenpath/hg19/bigZips/xenoMrna.fa.gz`, February 2009. [Accessed on 22-January-2015].

[Cra99] G.E. Crandall. Text file compression system utilizing word terminators, December 1999. US Patent 5,999,949.

[CRH11] John Wilkes Charles Reiss and Joseph L. Hellerstein. Google cluster-usage traces: format and scheme. `https://code.google.com/p/googleclusterdata/wiki/TraceVersion2`, November 2011. [Accessed on 28-July-2014].

[CW84] John G. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396–402, April 1984.

[CWC+15] Zhen Chen, Yuhao Wen, Junwei Cao, Wenxun Zheng, Jiahui Chang, Yinjun Wu, Ge Ma, Mourad Hakmaoui, and Guodong Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, 20(1):100–115, February 2015.

[CWT06] Hu Cao, Ouri Wolfson, and Goce Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, September 2006.

[CZS+11] Ming Chen, Hui Zhang, Ya-Yunn Su, Xiaorui Wang, Guofei Jiang, and K. Yoshihira. Effective vm sizing in virtualized data centers. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 594–601, May 2011.

[DAB+02] John R. Douceur, Atul Adya, William J. Bolosky, Daniel R. Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30, Microsoft Research, July 2002.

[DD13] Thomas H. Davenport and Jill Dyché. Big data in big companies. Report, Thomas H. Davenport and SAS Institute Inc., May 2013.

[Deu96] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DH13a] Dapeng Dong and J. Herbert. Efficient private cloud operation using proactive management service. In *International Journal of Cloud Computing (IJCC)*, pages 60–71, October 2013.

[DH13b] Dapeng Dong and J. Herbert. Energy efficient vm placement supported by data analytic service. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 648–655, May 2013.

[DH13c] Dapeng Dong and J. Herbert. A proactive cloud management architecture for private clouds. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 701–708, June 2013.

[DH14a] Dapeng Dong and J. Herbert. Content-aware partial compression for big textual data analysis acceleration. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 320–325, December 2014.

[DH14b] Dapeng Dong and J. Herbert. Fsaas: File system as a service. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 149–154, July 2014.

[DH15a] D. Dong and J. Herbert. Record-aware compression for big textual data analysis acceleration. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1183–1190, October 2015.

[DH15b] D. Dong and J. Herbert. Record-aware two-level compression for big textual data analysis acceleration. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16, November 2015.

[DHL92] R. Baeza-Yates Donna Harman, Edward Fox and W. Lee. *Information Retrieval: Data Structures and Algorithms, Inverted files*, chapter 3. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[DKS08] Anirban Dasgupta, Ravi Kumar, and Amit Sasturkar. De-duping urls via rewrite rules. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 186–194, New York, NY, USA, 2008. ACM.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[Dro] Dropbox. `https://www.dropbox.com/`. [Accessed on 20-September-2015].

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, March 1975.

[ES06] Andrea Esuli and Fabrizio Sebastiani. Sentiwordnet: A publicly available lexical resource for opinion mining. In *In Proceedings of the 5th Conference on Language Resources and Evaluation (LREC'06)*, pages 417–422, 2006.

[Fal85] Chris Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. *SIGMOD Rec.*, 14(4):63–82, May 1985.

[FDCD12] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *interactions*, 19(3):50–59, May 2012.

[Fen97] P. Fenwick. Symbol ranking text compressors. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 436–, March 1997.

[FK96] Aviezri S. Fraenkel and Shmuel T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64:31–55, 1996.

[FL08] Simone Faro and Thierry Lecroq. Efficient pattern matching on binary strings. *CoRR*, abs/0810.2390, 2008.

[FLPnSP09] Antonio Fariña, Susana Ladra, Oscar Pedreira, and Ángeles S. Places. Rank and select for succinct data structures. *Electronic Notes in Theoretical Computer Science*, 236:131 – 145, 2009. Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008).

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[FMMN04] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *In Proc.SPIREÕ04, LNCS 3246*, pages 150–160. Springer, 2004.

[FPST11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, April 2011.

[FR84] T. Ferguson and J. Rabinowitz. Self-synchronizing huffman codes (corresp.). *Information Theory, IEEE Transactions on*, 30(4):687–693, July 1984.

[Fra15] Mass Framingham. New idc forecast sees worldwide big data technology and services market growing to $48.6 billion in 2019, driven by wide adoption across industries. Report, IDC Research, Inc., November 2015. [Accessed on 12-November-2015].

[FSR12] Dan Feldman, Cynthia Sung, and Daniela Rus. The single pixel gps: Learning big data signals from tiny coresets. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 23–32, New York, NY, USA, 2012. ACM.

[FT98] Martin Farach and Mikkel Thorup. String matching in lempel-ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[FT13] Y. Fujiwara and V.D. Tonchev. High-rate self-synchronizing codes. *Information Theory, IEEE Transactions on*, 59(4):2328–2335, April 2013.

[GB14] Ramanathan Guha and Dan Brickley. RDF schema 1.1. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[GFE12] Kiem-Phong Vo Glenn Fowler, Landon Curt Noll and Donald Estlake. The fnv non-cryptographic hash algorithm. IETF, Network Working Group, September 2012.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[GGP12] H. Goudarzi, M. Ghasemazar, and M. Pedram. Sla-based optimization of power and migration cost in cloud computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 172 – 179, May 2012.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GMR96] D.W. Gillman, M. Mohtashemi, and R.L. Rivest. On breaking a huffman code. *Information Theory, IEEE Transactions on*, 42(3):972–976, May 1996.

[GNS$^+$06] Szymon Grabowski, Gonzalo Navarro, Ro Salinger, Rafal Przywarski, and Veli Mäkinen. A simple alphabet-independent FM-index. In *International Journal of Foundations of Computer Science (IJFCS)*, volume 17, pages 230–244, 2006.

[Gol66] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, July 1966.

[Goo] Google Drive. `https://www.google.com/drive/`. [Accessed on 20-September-2015].

[Gri14] Justin Grimmer. We are all social scientists now: How big data, machine learning, and causal inference work together. *PS: Political Science & Politics*, 48(01):80–83, dec 2014.

[Gro15] Vince Grolmusz. A note on the pagerank of undirected graphs. *Information Processing Letters*, 115(6–8):633 – 634, 2015.

[GSZS14] Oshini Goonetilleke, Timos Sellis, Xiuzhen Zhang, and Saket Sathe. Twitter analytics: A big data management perspective. *SIGKDD Explor. Newsl.*, 16(1):11–20, June 2014.

[Gzi] Gzip Compressor. `http://www.gzip.org/`. [Accessed on 12-January-2015].

[Has04] Reza Hashemian. Condensed table of huffman coding, a new approach to efficient decoding. *Communications, IEEE Transactions on*, 52(1):6–8, January 2004.

[HCG$^+$14] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

[HL90] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Commun. ACM*, 33(4):449–459, April 1990.

[Huf52] D.A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[IBMa] IBM BigInsights. `http://www-03.ibm.com/software/products/en/ibm-biginsights-for-apache-hadoop`. [Accessed on 26-August-2015].

[IBMb] IBM SmartCloud Enterprise. `https://www.ibm.com/cloud/enterprise`. [Accessed on 18-June-2013].

[IBY$^+$07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, June 2007.

[Igo15] Igor Pavlov. `http://www.7-zip.org/7z.html`, 2015. [Accessed on 19-August-2015].

[IM01] R.Y.K. Isal and A. Moffat. Word-based block-sorting text compression. In *Computer Science Conference, 2001. ACSC 2001. Proceedings. 24th Australasian*, pages 92–99, 2001.

[Int] Internet Archive. Stack exchange data dump. `https://archive.org/details/stackexchange`. [Accessed on 29-January-2015].

[ITU09] Recommendation ITU-R M.1677-1 International Morse Code. ITU-R recommendation, International Telecommunication Union, October 2009. M Series, Mobile, radiodetermination, amateur, and related satellite services.

[Jac88] Guy Joseph Jacobson. Succinct static data structures. 1988.

[Jac92] G. Jacobson. Random access in huffman-coded files. In *Data Compression Conference, 1992. DCC '92.*, pages 368–377, March 1992.

[JFAE12] Sang Woo Jun, K.E. Fleming, M. Adler, and J. Emer. Zip-io: Architecture for application-specific compression of big data. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 343–351, December 2012.

[JGL$^+$14] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, July 2014.

[JLH$^+$12] J.W. Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint vm placement and routing for data center traffic engineering. In *INFOCOM, 2012 Proceedings IEEE*, pages 2876–2880, March 2012.

[Joh15] George John. How artificial intelligence and big data created rocket fuel: A case study. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1629–1629, New York, NY, USA, 2015. ACM.

[JOSW10] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010.

[JPE+11] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 72–79, July 2011.

[JPHS10] Shirley Ann Jackson John P. Holdern, Eric Lander and Eric Schmidt. Report to the president and congress designing a digital future: Federally funded research and development in networking and information technology. PCAST WASHINGTON, D.C. 20502, The President's Council of Advisors on Science and Technology, December 2010.

[Kat12] U.N. Katugampola. A new technique for text data compression. In *Computer, Consumer and Control (IS3C), 2012 International Symposium on*, pages 405–409, June 2012.

[Ken13] Kenneth Neil Cukier and Viktor Mayer-Schoenberger . The rise of big data. `http://www.foreignaffairs.com/articles/139104/kenneth-neil-cukier-and-viktor-mayer-schoenberger/the-rise-of-big-data`, 2013. [Accessed on 20-April-2014].

[KJHMP77] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KKH+08] D. Kusic, J.O. Kephart, J.E. Hanson, N. Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. In *Autonomic Computing, 2008. ICAC '08. International Conference*, pages 3–12, June 2008.

[KLA+10] Hema Swetha Koppula, Krishna P. Leela, Amit Agarwal, Krishna Prasad Chitrapura, Sachin Garg, and Amit Sasturkar. Learning url patterns for webpage de-duplication. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 381–390, New York, NY, USA, 2010. ACM.

[KML13] Shamanth Kumar, Fred Morstatter, and Huan Liu. *Twitter Data Analytics*. Springer, 2013.

[KMMV02] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The vcdiff generic differencing and compression data format. RFC 3284, RFC Editor, June 2002.

[KRB13] Vince Kellen, Adam Recktenwald, and Stephen Burr. Applying big data in higher education: A case study. Data Insight & Social BI Executive Report 8, Cutter Consortium, December 2013.

[LAB$^+$09] Gavin LaRowe, Sumeet Ambre, John Burgoon, Weimao Ke, and Katy Börner. The scholarly database and its utility for scientometrics research. *Scientometrics*, 79(2):219–234, 2009.

[LAC$^+$11] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 961–972, New York, NY, USA, 2011. ACM.

[Lan01] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[LBK09] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 497–506, New York, NY, USA, 2009. ACM.

[lGA] Jean loup Gailly and Mark Adler. `http://www.gzip.org`. [Accessed on 20-May-2015].

[LHY12] Yih-Kai Lin, Shu-Chien Huang, and Cheng-Hsing Yang. A fast algorithm for huffman decoding based on a recursion huffman tree. *Journal of Systems and Software*, 85(4):974 – 980, 2012.

[LM99] N.J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference, 1999. Proceedings. DCC '99*, pages 296–305, March 1999.

[LML13] Himabindu Lakkaraju, Julian J. McAuley, and Jure Leskovec. What's in a name? understanding the interplay between titles, content, and communities in social media. In *ICWSM'13*", pages –1–1, 2013.

[LOOW14] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, January 2014.

[LQ14] Xiaolin Li and Judy Qiu, editors. *Cloud Computing for Data-Intensive Applications*. Springer, 1 edition, 2014.

[LR81] Jr. Langdon, G.G. and J. Rissanen. Compression of black-white images with arithmetic coding. *Communications, IEEE Transactions on*, 29(6):858–867, June 1981.

[LR13a] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: The twitter experience. *SIGKDD Explor. Newsl.*, 14(2):6–19, December 2013.

[LR13b] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: The twitter experience. *SIGKDD Explor. Newsl.*, 14(2):6–19, April 2013.

[LRW01] Geoffrey Leech, Paul Rayson, and Andrew Wilson. *Word Frequencies in Written and Spoken English: Based on the British National Corpus*. Routledge, 2001.

[LRW11] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 59–66, New York, NY, USA, 2011. ACM.

[LXJ$^+$11] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 171–182. ACM, 2011.

[Mah05] Matthew V Mahoney. Adaptive weighing of context models for lossless data compression. 2005.

[Man97] Udi Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inf. Syst.*, 15(2):124–136, April 1997.

[Mar79] G. N. N. Martin. Range encoding: an algorithm for removing redundancy for digitised message. In *Proceedings of the Video and Data Recording Conference*, Southampton, UK, July 1979.

[Mar15] Markus F.X.J. Oberhumer. LZO Compression. `http://www.oberhumer.com/opensource/lzo/`, 1996 - 2015. [Accessed on 01-October-2015].

[MCB$^+$11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[MGL$^+$11] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[ML13] Julian John McAuley and Jure Leskovec. From amateurs to connoisseurs: Modeling the evolution of user expertise through online reviews. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, pages 897–908. International World Wide Web Conferences Steering Committee, 2013.

[MM15] Stan Matwin and Jan Mielniczuk, editors. *Challenges in Computational Statistics and Data Mining*, volume 605 of *Studies in Computational Intelligence*. Springer, 2015.

[MMCJBB11] James Manyika, Brad Brown Michael Chui, Charles Roxburgh Jacques Bughin, Richard Dobbs, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Report, McKinsey Global Institute, May 2011.

[Mor15] Vincenzo Morabito. Big data and analytics for government innovation. In *Big Data and Analytics*, chapter Chapter 2: Big Data and Analytics for Government Innovation, pages 23–45. Springer Science & Business Media, 2015.

[MPI] MPI–Message Passing Interface. `http://www.mpi-forum.org/`. [Accessed on 18-June-2015].

[MPZ10] Xiaoqiao Meng, V. Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010.

[MS12] Donald Miner and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 1st edition, 2012.

[MT02] Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithms (The Springer International Series in Engineering and Computer Science)*. Springer, 2002.

[MVE$^+$14] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 1st edition, 2014.

[Nie11] Norman H Nie. The rise of big data spurs a revolution in big analytics. *Revolution Analytics Executive Briefing*, pages 1–8, 2011.

[NRNK10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, December 2010.

[OP15] C.E. Otero and A. Peter. Research directions for engineering big data analytics software. *Intelligent Systems, IEEE*, 30(1):13–19, January 2015.

[Ope] OpenMPI. `http://www.open-mpi.org/`. [Accessed on 18-June-2015].

[ORR$^+$13] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proc. VLDB Endow.*, 6(11):1092–1101, August 2013.

[PAF$^+$10] A. Parashar, M. Adler, K. Fleming, M. Pellauer, and J. Emer. Leap: A virtual platform architecture for fpgas. In *CARL'10: The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

[PB61] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, January 1961.

[PGK13] David Knott Peter Groves, Basel Kayyali and Steve Van Kulken. The big data revolution in healthcare: Accelerating value and innovation. Report, Center for US Health System Reform Bussiness Technology Office, January 2013.

[PP14] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Comput. Surv.*, 47(1):11:1–11:30, July 2014.

[PPR$^+$09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[Rai] RainStor. `http://rainstor.com/`. [Accessed on 20-September-2015].

[RAR15] RARLAB. `http://rarlab.com/`, 2002 - 2015. [Accessed on 05-September-2015].

[RD15] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, July 2015.

[RH15] Ron Shamir Rozov, Roye and Eran Halperin. Fast lossless compression via cascading bloom filters. *BMC Bioinformatics*, 15(13):S7, September 2015.

[Ris76] J.J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[RMW15] F. Rashid, A. Miri, and I. Woungang. Proof of storage for video deduplication in the cloud. In *Big Data (BigData Congress), 2015 IEEE International Congress on*, pages 499–505, June 2015.

[RQRSD14] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *The VLDB Journal*, 23(3):469–494, June 2014.

[Rus07] Philip Russom. BI Search and Text Analytics: New Addition to the BI Technology Stack. Technical report, The Data Warehousing Institute, Second Quarter 2007.

[RWZ$^+$15] R. Ranjan, L. Wang, A.Y. Zomaya, D. Georgakopoulos, X. Sun, and G. Wang. Recent advances in autonomic provisioning of big data applications on clouds. *Cloud Computing, IEEE Transactions on*, 3(2):101–104, April 2015.

[Sal08] David Salomon. *A Concise Introduction to Data Compression.* Undergraduate Topics in Computer Science. Springer, 2008.

[SD06] Dana Shapira and Ajay Daptardar. Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts. *Inf. Process. Manage.*, 42(2):429–439, March 2006.

[SdMNZBY00] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, April 2000.

[Sem97] *PODS '97: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 1997. ACM. 475970.

[Sew00] J. Seward. On the performance of bwt sorting algorithms. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 173–182, 2000.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. Technical Report 3, The Bell System Technical Journal, Fremont, CA, July 1948.

[Sha49] C. E. Shannon. The Transmission of Information. Technical Report 65, MIT Research Laboratory of Electronics, March 1949.

[SKA$^+$10] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. Classification of dna sequences using bloom filters. *Bioinformatics*, 26(13):1595–1600, 2010.

[SM10] Phil Sargeant and V Managing. Data centre transformation: How mature is your it? *Presentation by Managing VP, Gartner*, 2010.

[Sna] Snappy Compressor. `http://google.github.io/snappy/`. [Accessed on 02-August-2015].

[Spa] Spark MLlib. `http://spark.apache.org/docs/latest/mllib-guide.html`. [Accessed on 26-August-2015].

[Sta] Stack Sahara. `http://docs.openstack.org/developer/sahara/`. [Accessed on 26-August-2015].

[SW14] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2014.

[TB98] FionaJ. Tweedie and R.Harald Baayen. How variable may a constant be? measures of lexical richness in perspective. *Computers and the Humanities*, 32(5):323–352, 1998.

[The] The R Foundation. `http://www.r-project.org`. [Accessed on 02-August-2015].

[The13] The Open Group Base Specifications. IEEE Standard for Regular Expressions. *IEEE Std 1003.1*, 2013.

[Tit96] M.R. Titchener. Generalised t-codes: extended construction algorithm for self-synchronising codes. *Communications, IEE Proceedings-*, 143(3):122–128, June 1996.

[TSA$^+$10] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[Tuk77] John W Tukey. Exploratory data analysis. 1977.

[Twi] Twitter. Hadoop-lzo. `https://github.com/twitter/hadoop-lzo`. [Accessed on 28-February-2015].

[UMB10] Jacopo Urbani, Jason Maassen, and Henri Bal. Massive semantic web data compression with mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 795–802, New York, NY, USA, 2010. ACM.

[VAN08] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference* *on Middleware*, Middleware '08, pages 243–264. Springer-Verlag New York, Inc., 2008.

[VBVB09] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, pages 254–265. Springer-Verlag, 2009.

[Ven14] Rossano Venturini. *Compressed Data Structures for Strings on Searching and Extracting Strings from Compressed Textual Data*. Atlantis Press, 2014.

[Vit89] Jeffrey Scott Vitter. Algorithm 673: Dynamic huffman coding. *ACM Trans. Math. Softw.*, 15(2):158–167, June 1989.

[VTM14] John F. Gantz Vernon Turner, David Reinsel and Stephen Minton. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. Technical Report 1672, IDC Analyze the Future, April 2014.

[Wel84] T.A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[WH15] Dan Wang and Zhu Han. *Sublinear Algorithms for Big Data Applications (SpringerBriefs in Computer Science)*. Springer, 2015.

[Whi15] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 4th edition, 2015.

[Wika] Wikimedia Database Backup EN. `http://dumps.wikimedia.org/enwiki/enwiki-latest-abstract.xml`. [Accessed on 02-December-2014].

[Wikb] Wikimedia Database Backup ZH. `http://dumps.wikimedia.org/zhwiki/zhwiki-20140331-pages-articles-multistream.xml`. [Accessed on 20-January-2015].

[WM92] Sun Wu and Udi Manber. Fast text searching: Allowing errors. *Commun. ACM*, 35(10):83–91, October 1992.

[WM14] Lilian Weng and Filippo Menczer. Topicality and social impact: Diverse messages but focused messengers. Technical report, arXiv, 2014.

[WMB99] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 2nd edition, 1999.

[WVF15] PawełWoźniak, Robert Valton, and Morten Fjeld. Volvo single view of vehicle: Building a big data service from scratch in the automotive industry. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 671–678, New York, NY, USA, 2015. ACM.

[XF10] Jing Xu and J.A.B. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 179 – 188, December 2010.

[Yah06] Yahoo! Webscope Dataset. dataset ydata-ymusic-user-artist-ratings-v1.0. `http://research.yahoo.com/Academic_Relations`, 2002 - 2006. [Accessed on 20-April-2014].

[YLW$^+$13] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

[YP09] Hung-Chih Yang and D Stott Parker. Traverse: simplified indexing on large map-reduce-merge clusters. In *Database Systems for Advanced Applications*, pages 308–322. Springer, 2009.

[ZAW$^+$09] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 17–24, New York, NY, USA, 2009. ACM.

[ZCJW14] Zhi-Hua Zhou, Nitesh V. Chawla, Yaochu Jin, and Graham J. Williams. Big data opportunities and challenges: Discussions from data analytics perspectives [discussion forum]. *Comp. Intell. Mag.*, 9(4):62–74, November 2014.

[ZIP14] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1555–1566, New York, NY, USA, 2014. ACM.

[ZL77] J. Ziv and A Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, September 1978.

[ZM06] J. Zobel and A. Moffat. Inverted files for text search engines. *Computing Surveys*, 38:1–56, 2006.

[ZMR98] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, December 1998.

[ZYTC14] Hongbo Zou, Yongen Yu, Wei Tang, and H.M. Chen. Improving i/o performance with adaptive data compression for big data applications. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1228–1237, May 2014.

[ZZSY13] Xudong Zhang, W.X. Zhao, Dongdong Shan, and Hongfei Yan. Group-scheme: Simd-based compression algorithms for web text data. In *Big Data, 2013 IEEE International Conference on*, pages 525–530, October 2013.

# Abbreviations

**ACM** Association for Computing Machinery. 10

**AHC** Approximated Huffman Compression. i, ii, 9, 53–77, 135, A1, B2

**ARIMA** Auto Regressive Integrated Moving Average. E6

**ASCII** American Standard Code for Information Interchange. 18, 22, 24, 37, 41, 76, 100, 108

**BWT** Burrows-Wheeler Transform. 15, 27, 29

**CaC** Content-aware Compression. 8, 10, 132, 135–137

**CaPC** Content-aware Partial Compression. i, xi, 9, 36–52, 133, 135–137

**CD** Compressed Data. 104

**CRC** Cyclic Redundancy Check. 81

**CSV** Comma Separated Value. 5, 119

**DAG** Directed Acyclic Graph. 136

**DHCP** Dynamic Host Configuration Protocol. E7

**DI-Block** Default *Deflate* input block. 103

**DMC** Dynamic Markov-Chain. 21

**DNA** Deoxyribonucleic Acid. 55

**DNS** Domain Name Server. E7

**DO-Block** Default *Deflate* output data block. 103

**EMR** Amazon Elastic MapReduce. 3, E1

**ETS** Exponential sTate Space. E6

**Ext3** Linux Third Extended File System. 13

**F1PaBFD** One-step ahead Forecast-based Power-aware Best Fit Decreasing. E8

**FNV_1** Fowler–Noll–Vo (Version 1). 26

**FPGA** Field-Programmable Gate Array. 33

**FSaaS** File System as a Service. 10

**FTP** File Transfer Protocol. E7

**GA** Genetic Algorithm. E3

**GFS** Google File System. 136

**GPS** Global Positioning System. 32

**HDFS** Hadoop Distributed File System. 3, 11, 24, 33, 34, 78, 79, 101, 132, 134, 136

**HPC** High Performance Computing. 31

**HTTP** Hypertext Transfer Protocol. E7

**I/O** Input and Output. 4

**IaaS** Infrastructure as a Service. E7

**IEEE** Institute of Electrical and Electronics Engineers. 10

**IID** Independent Identically Distributed. E6

**IoT** Internet of Things. 2

**ISO** International Standards Organization. 101

**JSON** JavaScript Object Notation. 30, 36

**JVM** Java Virtual Machine. 120

**LiDAR** Light Detection And Ranging. 32

**LPSA** Lexical Permutation Sorting Algorithm. 15

**LZ** Lempel-Ziv 1977. 15

**LZMA** Lempel–Ziv Markov Chain. 15

**LZO** Lempel-Ziv-Oberhumer. 21

**MCMC** Markov chain Monte Carlo. 2

**MOA** Massive Online Analysis. 3

**mRNA** messenger Ribonucleic Acid. 70

**MSB** Most Significant Bit. 24, 85

**MTF** Move-To-Front. 16

**NoSQL** Not Only Structured Query Language. 31, 136

**NTFS** Microsoft® New Technology File System. 13

**ORC** Optimized Record Columnar. 30

**PABFD** Power Aware Best Fit Decreasing. E3

**PPM** Prediction by Partial Matching. 20

**QFS** Quantcast File System. 136

**QoS** Quality of Service. E2

**RaC** Record-aware Compression. ii, 9, 78–96, 134, 135, A1

**RaPC** Record-aware layerd Partial Compression. ii, 9, 97–131, 135, A1

**RAR** Roshal Archive compRessed. 21

**RDF** Resource Description Framework. 32

**rDSS** The *R* Decision Support System. E2

**RLE** Run-Length Encoding. 28

**RNA** Ribonucleic Acid. 55

**RPM** Revolutions Per Minute. 126

**RW** Random Walk. E6

**SLA** Service Level Agreement. E1

**SSE** Streaming SIMD (Single Instruction Multiple Data) Extensions. 33

**STS** Structural Time Series. E6

**TCP/IP** Transmission Control Protocol/Internet Protocol. E11

**TSV** Tab Separated Value. 119

**URL** Uniform Resource Locator. 32, 109

**UTF-8** Unicode Transformation Format (8-Bit). 24, 34, 37, 41

**VM** Virtual Machine. E1

**W3C** World Wide Web Consortium. 32

**XML** eXtensible Markup Language. 5, 30, 36