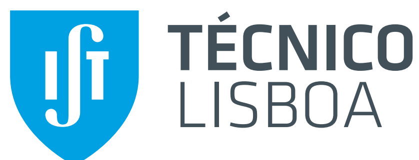

Efficient Support for Selective MapReduce Queries

Manuel da Silva Santos Gomes Ferreira

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Advisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member:	Prof. Nuno Preguiça

October 2014

Acknowledgements

I would like to express my special thanks to my advisor Professor Luís Rodrigues who gave me the opportunity to do this thesis and also the vital guidance and motivation.

I extend my sincere gratitude to João Paiva for taking part in useful decisions and giving fruitful advices during the execution of all the work and preparation of this thesis. A special thanks also to my family and friends for their support during this thesis.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the project PEPITA (PTDC/EEI-SCR/2776/2012) and via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013.

Lisboa, October 2014

Manuel da Silva Santos Gomes Ferreira

For my parents,

Resumo

Atualmente, existe uma crescente necessidade de analisar grandes volumes de dados, uma tarefa que requer infraestruturas de computação e armazenamento especializadas. O paradigma MapReduce tem-se tornado fundamental para paralelizar computações complexas sobre grandes volumes de dados. Conhecido pela sua escalabilidade, fácil utilização e tolerância a faltas, o MapReduce tem sido extensamente utilizado por aplicações de diferentes domínios.

O trabalho descrito nesta tese propõe e avalia o ShortMap, um sistema destinado a suportar eficientemente pesquisas MapReduce que necessitam de processar apenas um sub-conjunto de todos os dados. O sistema proposto recorre a um formato de dados apropriado para suportar pesquisas seletivas e mecanismos de indexação de forma a melhorar a rapidez de acesso aos dados, encurtando significativamente a fase Map das execuções. Uma extensa avaliação experimental do ShortMap mostra que, ao evitar ler blocos irrelevantes, a nossa solução permite atingir uma melhoria de até 80 vezes quando comparada com a distribuição atual do Hadoop. Para além disso, o nosso sistema supera também outras concretizações do MapReduce que recorrem a variantes das técnicas integradas no ShortMap. O ShortMap é de código-fonte aberto e está disponível para ser descarregado.

Abstract

Today, there is an increasing need to analyse very large datasets, that have been coined "big data", and that require specialised storage and processing infrastructures. MapReduce is a programming model aimed at processing big data in a parallel and distributed manner. Known for its scalability, ease of use and fault-tolerance, MapReduce has been widely used by different domain applications.

The work described in this thesis proposes and evaluates ShortMap, a system that relies on a combination of techniques aimed at efficiently supporting selective MapReduce jobs that are only concerned with a subset of the entire dataset. We combine the use of an appropriate data layout with data indexing tools to improve the data access speed and significantly shorten the Map phase of the jobs. An extensive experimental evaluation of ShortMap shows that, by avoiding reading irrelevant blocks, it can provide speedups up to 80 times when compared to the basic Hadoop implementation. Further, our system also outperforms other MapReduce implementations that use variants of the techniques we have embedded in ShortMap. ShortMap is open source and available for download.

Palavras Chave
Keywords

Palavras Chave

Grandes volumes de dados

Acesso aos dados

MapReduce

Fase Map

Desempenho

Keywords

Big data

Data access

MapReduce

Map phase

Performance

Contents

1 Introduction **3**

1.1 Motivation 4

1.2 Contributions 6

1.3 Results 6

1.4 Research History 6

1.5 Structure of the Document 7

2 Related Work **9**

2.1 Introduction 9

2.2 MapReduce Basics 9

2.2.1 Mappers and Reducers 10

2.2.2 MapReduce Steps 11

2.2.3 Data Locality in MapReduce 13

2.3 A MapReduce Implementation: Hadoop 13

2.3.1 Storage Layer 14

2.3.2 Processing Layer 14

2.4 Selective Queries 15

2.4.1 Word Counting 15

2.4.2 Selective Query 16

2.5 MapReduce Performance 16

2.5.1 Indexing 17
2.5.2 Data Layout 18
2.5.3 Co-location 21
2.6 Relevant Systems 21
2.6.1 Hadoop++ 21
2.6.2 LIAH 23
2.6.3 E3 24
2.6.4 Full-Text Indexing 26
2.6.5 Trojan Data Layouts 28
2.6.6 RCFile 29
2.6.7 CIF Storage Format 31
2.6.8 Comparative Analysis 33

3 ShortMap **37**

3.1 Introduction 37
3.2 Data Layout 37
3.3 Data Grouping 38
3.4 Indexing 39
3.5 Other ShortMap Aspects 40
3.5.1 Replication 41
3.5.2 Index Storage 41
3.5.3 Block Transfer Pre-Validation 41
3.5.4 Pre-Processing 42
3.6 ShortMap Implementation 43
3.6.1 Hadoop Brief Overview 43

3.6.2 Most Important Tweaks on Hadoop 44

4 Evaluation **47**

4.1 Introduction 47

4.2 Index Structures Micro-benchmark 49

4.3 ShortMap against Indexed Row-Oriented Stores 50

4.3.1 MapReduce Performance 51

4.3.2 Index Space Usage 52

4.4 ShortMap against Column-Oriented Stores 53

4.5 ShortMap against Indexed Column-Oriented Stores. 54

4.6 ShortMap against Hadoop 56

4.7 ShortMap against Spark 57

5 Conclusions **61**

5.1 Conclusions 61

5.2 Future Work 61

Bibliography **66**

List of Figures

1.1 Cost of reading data 4

1.2 Distribution of values in a realistic dataset 5

2.1 MapReduce steps 11

2.2 Row-oriented data layout 19

2.3 Column-oriented data layout 20

2.4 Column groups data layout 20

2.5 Pax data layout 21

2.6 Indexed data layout 22

2.7 Range index 25

2.8 Per-replica Trojan Layout 28

2.9 Data layout of RCFile 30

2.10 Data layout of CIF 31

2.11 Co-location with CIF 31

2.12 Skip list 32

3.1 Row groups in ShortMap 39

4.1 Cost of reading data depending on job type and query selectivity 48

4.2 Read operation speedup depending on the index being used by ShortMap 49

4.3 Entries read by Hadoop, Hadoop++ and ShortMap 50

4.4 Bytes read by Hadoop, Hadoop++ and ShortMap 50

4.5 Speedup of ShortMap over the Hadoop++ 51

4.6 Entries read by Hadoop, CIF and ShortMap 53

4.7 Bytes read by Hadoop, CIF and ShortMap 53

4.8 Speedup of ShortMap against CIF 53

4.9 Speedup of ShortMap against an indexed column-oriented store 54

4.10 Bytes read by Hadoop and ShortMap 56

4.11 Entries read by Hadoop and ShortMap 56

4.12 Speedup of ShortMap over the unmodified Hadoop 56

4.13 Speedup of ShortMap against Spark and Hadoop for the word counting job . . . 58

4.14 Speedup of ShortMap against Spark and Hadoop for the sentiment analysis job . 58

List of Tables

2.1 Trojan Data Layout - Access pattern 29

2.2 State of the art systems 34

4.1 Space usage of Per-Node Index and Per-Block Index 52

Acronyms

DBMSs Database Management Systems

GFS Google File System

HDFS Hadoop Distributed File System

UDFs User Defined Functions

LIAH Lazy Indexing and Adaptivity in Hadoop

TLW Trojan Layout Wizard

RCFile Record Columnar File

RDDs Resilient Distributed Datasets

1 Introduction

Today, there is an increasing need to analyse very large datasets, a task that requires specialised storage and processing infrastructures. The problems associated with the management of very large datasets have been coined “big data”. Big data requires the use of massively parallel software, running on hundreds of servers, in order to produce useful results in reasonable time.

MapReduce is a programming model aimed at processing big data in a parallel and distributed manner. Known for its scalability, ease of use and fault-tolerance, MapReduce has been widely used by different domain applications such as search engines, social networks, or processing of scientific data. The MapReduce framework shields programmers from the problems of writing a distributed program, such as managing the fault-tolerance or the explicitly programming of the communication among processes. Users only have to define their programs mainly by means of two simple functions called *Map* and *Reduce*. The run-time automatically assigns Map and Reduce tasks to nodes, manages the communication among those nodes, monitors the execution of the tasks, and adapts to the size of the cluster, making the application easily scalable.

Originally, MapReduce was designed for jobs such as web indexing, that need to process the entire dataset (Page, Brin, Motwani, & Winograd 1999). However, as the range of applications of MapReduce grows, it is frequently used to execute queries that are only concerned with a small fraction of the entire dataset (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010; Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012). For instance, communications service providers maintain datasets about their customers that they can use as an additional source of revenue by selling analysis of data to third parties for market research (van der Lande 2013). In such applications, data analytics may be performed on demand, for the specific entries that are of the interest of each customer.

MapReduce is very flexible, and does not enforce any particular data model or format.

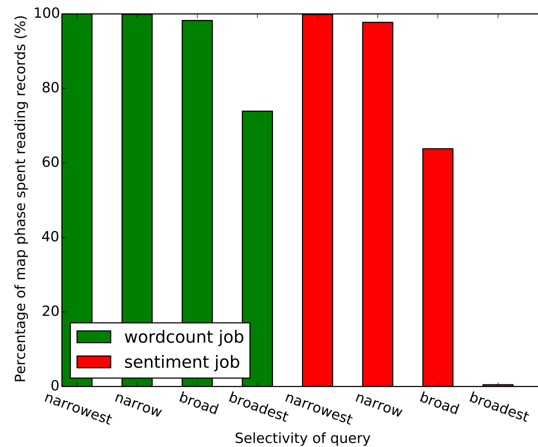

Figure 1.1: Cost of reading data, relative to the duration of Map tasks, depending on job type and query selectivity. The "broadest" query filters records by the most common value in the dataset, while the "narrowest" query filters by the most uncommon one.

However, being completely oblivious to the data format, MapReduce is generally forced to scan the entire input, even if only fractions of it are relevant for a particular job. Due to these limitations, MapReduce is far from reaching the performance of well-configured parallel Database Management Systems (DBMSs) (Pavlo, Paulson, Rasin, Abadi, DeWitt, Madden, & Stonebraker 2009), which on the other hand, are much more difficult to tune and to configure and requires the data to be fit into a rigid and well-define schema.

1.1 Motivation

Current MapReduce implementations are not well tailored to support selective operations that touch only in a subset of the data. According to the MapReduce model, it is the role of the Map tasks to select the appropriate entries of the dataset, that are relevant to the computation being performed, and pass those entries to Reducers. To perform this selection, the Map tasks may be forced to read the entire dataset, even if only a small fraction is relevant for the query being performed. This can consume a significant fraction of the entire MapReduce job. Figure 1.1 illustrates the cost of reading the input data, in comparison with the cost of the complete Map phase, for two types of jobs and two types of selective queries when using the default Hadoop implementation over a real Twitter dataset.

The "wordcount" job is a common example of an Hadoop workload, and consists of counting the frequency of words in the text; the "sentiment" job consists of calculating the sentiment of

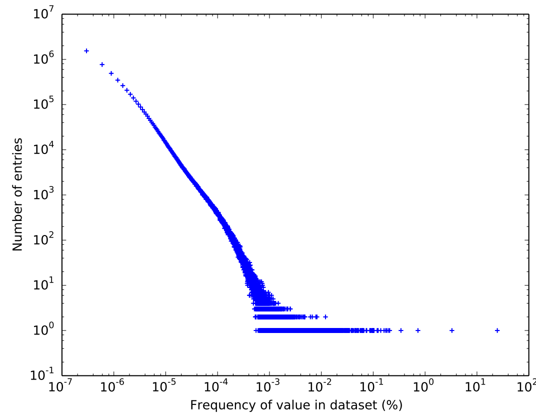

Figure 1.2: Distribution of the "user location" values in a realistic Twitter dataset.

twitter users, a realistic workload which mimics the big data processing required to extract scientific results from Twitter datasets (Kloumann, Danforth, Harris, Bliss, & Dodds 2012; Mitchell, Frank, Harris, Dodds, & Danforth 2013; Frank, Mitchell, Dodds, & Danforth 2013; Bertrand, Bialik, Virdee, Gros, & Bar-Yam 2013). As it can be seen, when querying for all but the most common item, the Map phase can take from 60% to 99% of the total querying time. In the case of the most common item, since it is a narrow search, most of the time is invested in processing the data. In fact, previous research has already identified several Map-heavy workloads which justify Map task optimization as an interesting research area (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010; Palanisamy, Singh, Liu, & Jain 2011; Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012; Richter, Quiané-Ruiz, Schuh, & Dittrich 2012; Kavulya, Tan, Gandhi, & Narasimhan 2010; Zaharia, Konwinski, Joseph, Katz, & Stoica 2008; Lin, Zhang, Wierman, & Tan 2013).

A common characteristic of many big data datasets, that motivates the design of our work, is that the distribution of the frequencies of values for any attribute typically follows a highly skewed distribution such as a Zipfian distribution (Zipf 1935). This is illustrated in Figure 1.2, that presents the distribution of frequencies of the "user location" attribute in a sample of the real Twitter dataset used to evaluate ShortMap. This kind of distribution is particularly amenable for being indexed since the most values are uncommon. This suggests that, an index for these values will have the capability of selecting only small fractions of the data for each value, achieving efficient queries.

1.2 Contributions

This work addresses the problem of data placement in MapReduce. More precisely, the thesis analyzes, implements and evaluates techniques to improve the data access speed and significantly shorten the Map phase of MapReduce jobs that are only concerned with a subset of the entire dataset. As a result, the thesis makes the following contributions:

- A novelty combination of a per-node index with data grouping that improves selective queries in MapReduce.
- The architecture of a system that incorporates the use of an appropriate data layout with data grouping and indexing tools.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A publicly available prototype implementation of the described architecture.
- An experimental evaluation of the implemented prototype, which can achieve speedups of up to 80 times when compared with a basic MapReduce implementation.

1.4 Research History

In the beginning, the main focus of this work was to study the state of the art solutions specifically concerned with optimizing the Map phase of selective workloads. As a result of that study this work would produce a new system that would improve the existent ones. During the bibliographic research, the indexing capability already existent in DBMSs was integrated into MapReduce (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010; Richter, Quiané-Ruiz, Schuh, & Dittrich 2012). This index reduces the data that needs to be analyzed during a job by reading from disk only the records that satisfy the query at hand. Also, a columnar store (Floratou, Patel, Shekita, & Tata 2011) keeps each data attribute separately in disk in order to select only those referred by a given query, avoiding to read all the attributes from disk

when unnecessary. This way, the main idea behind this work emerged from the combination of these two main techniques. However, we found that without an appropriate data grouping columnar indexes may be quite ineffective.

During my work, I benefited from the fruitful collaboration of João Paiva.

Previous descriptions of this work were published in (Ferreira, Paiva, & Rodrigues 2014).

1.5 Structure of the Document

The rest of this document is organized as follows. For self-containment, Section 2 provides an introduction to MapReduce and a description of different systems previous to this work. Chapter 3 describes the architecture and the algorithms used by ShortMap and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

2.1 Introduction

In order to improve the performance of MapReduce, several research projects have experimented to augment the basic MapReduce with additional mechanisms that can optimize data access on the Map phase of selective MapReduce jobs. An example of one such mechanism is the use of indexes, a technique that has proven to be useful in the context of DBMSs, to reduce the scope of a search. In this chapter we introduce MapReduce and survey the most relevant optimisations that have been proposed.

2.2 MapReduce Basics

MapReduce (Dean & Ghemawat 2004) is a framework for supporting the parallel processing of large amounts of unstructured data. MapReduce includes both a model to structure the computation and a runtime to parallelize the computation on a cluster of workstations in a fault-tolerant manner.

The computation model supported by MapReduce is quite generic. Data processing must be composed into two functions that are executed sequentially, namely: a *Map* and a *Reduce* function. Basically, the Map function is responsible for processing parts of the input and generating intermediate results which are then collected and combined on the Reduce function to generate the final output. This allows a broad range of computations to be modelled as MapReduce jobs, which is one of the reasons why it is a widely used framework, applied on diverse types of unstructured data such as documents, industry logs, and other non-relational data. In fact, MapReduce was originally proposed by Google but has been adopted by many different applications that have to manage large volumes of data, such as Facebook (Aiyer, Bautin, Chen, Damania, Khemani, Muthukkaruppan, Ranganathan, Spiegelberg, Tang, & Vaidya 2012)

or scientific processing of the outputs produced by particles accelerators.

A runtime supporting the MapReduce model in a cluster of workstations automatically assigns Map and Reduce tasks to hosts, monitors the execution of those tasks, and relaunches tasks if failures occur (for instance, if a host fails). This relieves the programmer from being concerned with the control plane during the execution of the application. The runtime can adapt the job scheduling to the number of servers existing in the cluster, making MapReduce jobs easily scalable.

2.2.1 Mappers and Reducers

A program running on MapReduce is called a *job*, and it is composed by a set of *tasks*. Some of these tasks apply the Map function to the input (the Map tasks), while others apply the Reduce function to the intermediate results (the Reduce tasks). So, a task can be viewed as an indivisible computation run in a node of the cluster, which operates over a portion of data. Nodes that compute tasks in the Map phase are called *Mappers* whereas the ones performing tasks in the Reduce phase are named *Reducers*.

When a job is submitted, the Map tasks are created and each one is associated a portion of the input data. The Map function takes as input a key/value pair and, after some computation, it generates an intermediate key/value pair. The intermediate key/value pairs generated by these tasks are then assigned to Reducers. This means that, during the execution, the intermediate results generated by Mappers must be sent to the Reducers (typically, via the network), such that a Reducer will get all intermediate key/value pairs that share the same intermediate key.

The Reduce tasks are then responsible for applying the Reduce function to the intermediate values of all keys assigned to it. This function will combine all the intermediate values in order to generate the final result of a job.

The nodes that execute Map or Reduce tasks are named *workers*, since they are the nodes that are responsible for doing the computational work. Each worker may serve as Mapper, Reducer, or as both if it performs Map and Reduce tasks. In addition to workers, there is one special node called *master* whose responsibility is to perform the task scheduling, i.e. the assignment of tasks to workers, and also to ensure the job completion (by monitoring the workers and re-assigning tasks to nodes if needed).

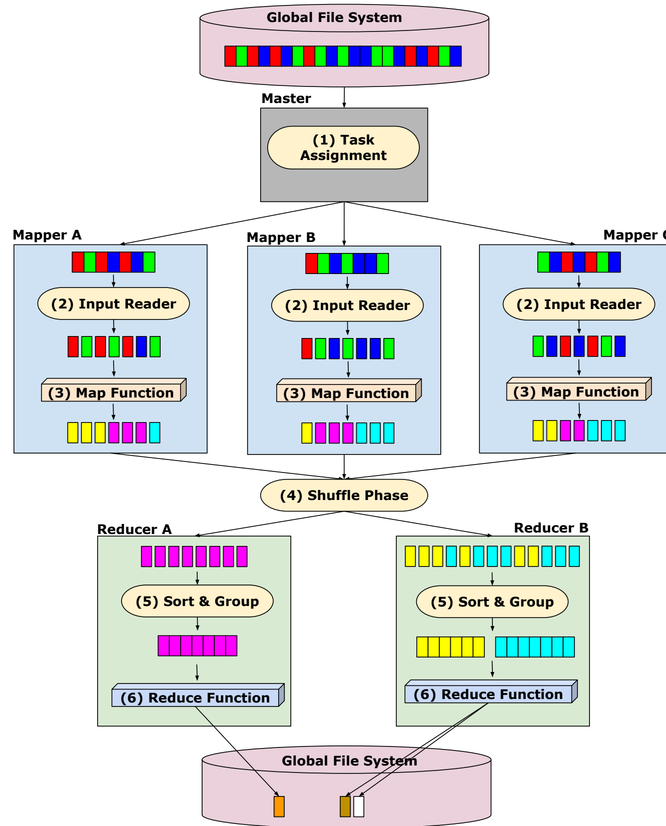

Figure 2.1: MapReduce Steps

2.2.2 MapReduce Steps

Upon job submission, the input data is automatically partitioned by the MapReduce library into pieces, called splits. Each Map task processes a single split, thus $\mathcal{M}$ splits entails $\mathcal{M}$ Map tasks.

The Map phase consists in having Mappers read the corresponding splits and generate intermediate key/value pairs. Typically, the Mapper uses a helper *input reader* to read the raw input data. The goal of the reader is to convert the input to a key/value pairs amenable to be processed by the Map function. This allows the MapReduce framework to support a variety of jobs that are not tied to any specific input format.

After the Map phase, a partition function is applied over the intermediate key space to divide it into $\mathcal{R}$ partitions, each one to be processed in a Reduce task (e.g., $hash(key)$ **mod** $\mathcal{R}$). Both $\mathcal{R}$ and the partitioning function can be configured by the user.

Figure 2.1 illustrates the sequence of steps executed during the execution of a a MapReduce job, which are enumerated below:

1. **Task assignment**: The input data is divided into splits and, for each split, the master creates a Map task and assigns it to a worker.

2. **Input Reader**: Each Map task executes a function to extract the key/value pairs from the raw data inside the splits.

3. **Map function**: Each key/value pair is fed into the user-defined Map function that can generate zero, one or more intermediate key/value pairs. Since these pairs represent temporary results, the intermediate data is stored in the local disks of the Mappers.

4. **Combiner phase (optional)**: If the user-defined Reduce function is commutative and associative, and there is a significant repetition of intermediate keys produced by each Mapper, an optional user-defined Combiner function can be applied to the outputs of the Mapper (Dean & Ghemawat 2004). The goal is to group, according to the intermediate key, all the intermediate values that were generated by the Map tasks of each Mapper. Typically, the same Reduce function code is used for the Combiner function. If this Combiner function is applied, each Mapper outputs only one intermediate value per intermediate key.

5. **Shuffle phase**: The intermediate key/value pairs are assigned to Reducers by means of a partition function in a manner such that all intermediate key/value pairs with the same intermediate key will be processed by the same Reduce task and hence by the same Reducer. Since these intermediate key/value pairs are spread arbitrarily across the cluster, the master passes to Reducers the information about the Mappers's location, so that each Reducer may be able to remotely read its input.

6. **Sort & Group**: When the remote reads are finished, each Reducer sorts the intermediate data in order to group all its input pairs by their intermediate keys.

7. **Reduce function**: Each Reducer passes the assigned intermediate keys, and the corresponding set of intermediate values, to the user-defined Reduce function. The output of the Reducers is stored in the global file system for durability.

During this entire sequence of steps, the master periodically pings the workers in order to provide programmer-transparent fault-tolerance. If no response is obtained within a given interval, the worker is considered as failed and the Map or Reduce tasks running on that node are restarted. If the master fails, the whole MapReduce computation is aborted (Dean & Ghemawat 2004).

2.2.3 Data Locality in MapReduce

MapReduce makes no assumptions on how the computation layer is related with the storage layer, as they may be completely independent. However, in practice, most MapReduce implementations rely in a distributed file system (Google File System (GFS) (Ghemawat, Gobioff, & Leung 2003) for Google's MapReduce (Dean & Ghemawat 2004), or Hadoop Distributed File System (HDFS) (Shvachko, Kuang, Radia, & Chansler 2010) with Hadoop MapReduce (http://hadoop.apache.org)). Furthermore, often, the nodes responsible for the computational work may be the same nodes that store the input data.

In this common configuration, it is possible to save network bandwidth if the master takes into account which data each node is storing when scheduling the Map tasks. First, it attempts to schedule a Map task on a node that stores the data to be processed by that Map task. If this is not possible, then it will attempt to schedule the task on the closest node (network-wise) to the one storing the data.

2.3 A MapReduce Implementation: Hadoop

Apache Hadoop (http://hadoop.apache.org) is the most popular open-source implementation of the MapReduce framework. It is written in Java and has received contributions from large companies such as Facebook and others (Aiyer, Bautin, Chen, Damania, Khemani, Muthukkaruppan, Ranganathan, Spiegelberg, Tang, & Vaidya 2012). Similarly to Google's MapReduce, Hadoop also employs two different layers: the HDFS, a distributed storage layer responsible for persistently storing the data among the cluster nodes; and the Hadoop MapReduce Framework, a processing layer responsible for running MapReduce jobs.

2.3.1 Storage Layer

Hadoop DFS (HDFS) is a distributed file system designed to store immutable files, i.e., once the data is written into the HDFS is not modifiable but can be read multiple-times.

The HFDS implementation uses three different entities: one NameNode, one SecondaryNameNode and one or more DataNodes. A NameNode is responsible for storing the metadata of all files in the distributed file system. In order to recover the metadata files in case of a NameNode failure, the SecondaryNameNode keeps a copy of the latest checkpoint of the filesystem metadata.

Each file in HDFS is divided into several fixed-size blocks (typically configured with 64MB each), such that each block is stored on any of the DataNodes. Hadoop replicates each block (by default, it creates 3 replicas) and places them strategically in order to improve availability: two replicas on DataNodes on the same rack and the third one on a different rack, to prevent from loss of data, should an entire rack fail.

It is worth noting the difference between input splits and HDFS blocks. While an HDFS block is an indivisible part of a file that is stored in each node, an input split is logical data that is processed by a Map task. There is no requirement for splits to be tied to blocks, and not even files. Consider the case where a file is split by lines, such that each Map task processes one line (Hadoop supports the usage of arbitrary split functions). It may happen that a line overflows from an HDFS block, i.e., the split boundaries do not coincide with the HDFS block boundaries. In this case, the Mapper processing that line must retrieve (possibly from some other node) the next HDFS block to obtain the final part of the line.

2.3.2 Processing Layer

The entities involved in the processing layer are one master, named the JobTracker, and one or more workers, named the TaskTrackers. The role of the JobTracker is to coordinate all the jobs running on the system and to assign tasks to run on the TaskTrackers which periodically report to the JobTracker the progress of their running tasks.

Hadoop's default scheduler makes each job use the whole cluster and takes the jobs' priorities into account when scheduling them. This means that higher priority jobs will run first. However,

Hadoop also supports other schedulers, including shared cluster schedulers that allow running jobs from multiple users at the same time.

In terms of task scheduling, Hadoop does not build an *a priori* plan to establish which tasks are going to run on which nodes; instead Hadoop decides on which nodes to deploy each task in runtime (Lee, Lee, Choi, Chung, & Moon 2012). As soon as a worker finishes a task it is assigned the next one. This means that should a worker finish all tasks associated with the data it stores locally, it may be fed tasks which entail obtaining data from other workers.

There are ten different places in the query-execution pipeline of Hadoop where User Defined Functions (UDFs) may be injected (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010). A user of the framework can define how the input data is split and how a split is parsed into key/value pairs, for example. This aspect makes Hadoop an easily customizable MapReduce framework.

2.4 Selective Queries

We now provide two different examples of applications of MapReduce, each of which has different requirements in terms of data management. The first is a basic MapReduce example, that is interested in analyzing all the input data. The second application requires only a subset of the data to produce the final result.

2.4.1 Word Counting

The word counting problem consists in counting the number of occurrences of the different words that appear in a text file. Several real world problems, like log analysis or text mining, can be viewed as instances of the word counting problem.

When MapReduce is used to solve this problem, one first starts by splitting the input text file and then assigning the resulting splits to the Mappers. For each occurrence of *word*, the Map function emits the intermediate key/value pair in the form of $\langle word, 1\rangle$. At the end of this phase, each Mapper outputs as many intermediate key/value pairs as the number of occurrences of all the words it has processed, and those pairs are then transferred from the Mappers to the Reducers in the Shuffle phase.

The assignment of keys to Reducers is typically performed with the help of a hash function. Each Reducer will get the pairs corresponding to all words assigned to it by the hash function. The Reduce function then groups all pairs by word, and outputs pairs in the format $\langle word, \text{sum} \rangle$, where *sum* corresponds to the sum of the values associated with *word* in the intermediate results.

This use case is one which perfectly fits the usage of the Combiner function. In order to reduce the amount of data transferred in the Shuffle phase, the Mappers can do part of the Reduce operation: each one sums the occurrences for each word that has identified. When using the Combiner step, the number of intermediate key/value pairs output by each Mapper would be equal to the number of different words it has processed.

Overall, the MapReduce model has a good fit to the word counting problem, since in this case both Map and Reduce functions do useful work in a balanced manner.

2.4.2 Selective Query

The selection problem consists in returning only the set of records that meet some condition to produce the result. It is analogous to the SELECT operation in traditional DBMSs.

The first step to process such jobs, as normally happens in MapReduce, is to split the input dataset among the Mappers that will extract the records from splits and pass them to the Map function. Typically, in this sort of problem, the Map function is responsible for doing the filtering, i.e. perform the test condition on each record and passing to the Reducers only those that satisfy it. The Reduce function is then applied over the selected records.

In general, this type of problem is characterised by a longer and heavier mapping step, given that Mappers need to analyse all the input, a task that is bounded by the I/O time required to read all the data from disk (Pavlo, Paulson, Rasin, Abadi, DeWitt, Madden, & Stonebraker 2009; Jiang, Ooi, Shi, & Wu 2010).

2.5 MapReduce Performance

MapReduce may not provide the desired performance in some use cases, such as the selection scenario presented in the previous section. This fact has motivated several recent works, which aim at improving MapReduce performance for different usage patterns. In this section we

describe the most relevant of these techniques, such as indexing, data layouts, and co-location, which aim at improving data accesses during the Map phase of selective query processing.

2.5.1 Indexing

In a plain MapReduce framework, the execution of a job requires that a full scan of the dataset is performed (i.e., by reading all the records one by one) even if only a small subset of them is going to be selected. Furthermore, all splits generate Map tasks even if some of them do not contain relevant data and will not create any intermediate data. Also, in the Map phase, all blocks have to be completely fetched from disk, which implies that all attributes of all records are brought to main memory without taking into account which ones the job is interested in. This problem can be avoided if the input data is indexed before the mapping step takes place.

Shortly, an index is built for one attribute and consists in a list of different values of that attribute contained in the indexed data, as well as the positions where these values can be found in the input file. Consequently, an index can significantly reduce the amount of data that must be read from disk, leading to a better performance of jobs, such as the ones implementing selection, where only a subset of the records is actually needed to produce the desired output.

Some systems create an unique index based in a single attribute (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010), while others consider multiple attributes which leads to building multiple indexes (Richter, Quiané-Ruiz, Schuh, & Dittrich 2012; Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012; Lin, Ryaboy, & Weil 2011).

The following indexing techniques can be applied:

- **Record-level indexing**: generally, this approach requires having the index inside each split, which is then used at query time to select the relevant records from the split, instead of reading the entire split record by record (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010).
- **Split-level indexing**: it has been observed that the execution time of a query depends on the number of waves of Map tasks that are executed, which in turn depends on the number of processed splits (Eltabakh, Özcan, Sismanis, Haas, Pirahesh, & Vondrak 2013). Furthermore, the time to process a split is dominated by the I/O time for reading its data

plus the overhead of launching the Map task to process it. So, in the end, these authors support that the time taken to process a small number of records in a split can be roughly the same as the time required to process all records of that split. On Section4.2, we present a micro-benchmark that we ran to verify this statement where we compare a record with a split-level index. As we will see, the effectiveness of a split-level index was not observed which means that it was worth the use of a record-level index. The idea of split-level indexes is to keep, for each attribute value, information about the splits that contain at least one record with that attribute value, such that splits with no relevant information are not processed. When processing a query, the master uses this index in order to get the minimal set of splits needed for that query, so Map tasks are generated only for these splits.

- **Block-level indexing**: in cases where splits are comprised of multiple blocks, one can create block-level indexes. Since blocks contain multiple records, these indexes have the potential to bring meaningful time savings, according to the observations of (Eltabakh, Özcan, Sismanis, Haas, Pirahesh, & Vondrak 2013)). Block-level indexes map attribute values to blocks that contain at least one record with that attribute value. Through these indexes, when processing a split, the relevant blocks are loaded, (possibly decompressed), and processed; whereas the blocks known not to match the selection criteria are skipped.

2.5.2 Data Layout

As mentioned in Section 2.3.1, the storage unit of HDFS consists in a fixed-size (typically 64MB) block of data. In fact, the way the data is organized *within* this block of data can greatly affect the response times of the system. We call this internal organization of blocks the *data layout*. We now introduce four different data layouts that may be applied on MapReduce:

- **Row-oriented**: As Figure 2.2 shows, when using this layout all fields of one record are stored sequentially in the file, and multiple records are placed contiguously in the disk blocks. This layout provides fast data loading since, in most cases, the data to be stored is supplied row-by-row, i.e. one entire record at a time. Row-based systems are designed to efficiently process queries where many columns of a record need to be processed at the same time, since an entire row can be retrieved with a single access to disk. On the other

hand, it does not perform so well when supporting queries that only need to look at a small subset of columns. Furthermore, since each row typically contains different attribute from different domains, a row often has high *information entropy*, which makes this data layout not well suited for compression (Abadi, Madden, & Hachem 2008).

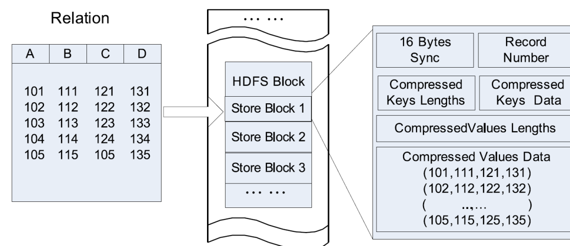

Figure 2.2: Row-oriented data layout (He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011)

- **Column-oriented**: When using this layout, datasets are vertically partitioned by column, each column is stored separately and accessed only when the corresponding attribute is needed (see Figure 2.3). This layout is well suited for read operations that accessed a small number of columns, since columns that are not relevant for the intended output are not read. Furthermore, it can achieve a high compression ratio since the data domain is the same in each column, hence each column tends to exhibit low *information entropy* (Abadi, Madden, & Hachem 2008). The better the data is compressed, the shorter the I/O time spent in reading it from disk. In this data format, each column is stored in (one or more) HDFS block, which means that it may be impossible to guarantee that all columns will be stored on the same node. As a result, record reconstruction may require network transfers to fetch the needed columns from different storage nodes.

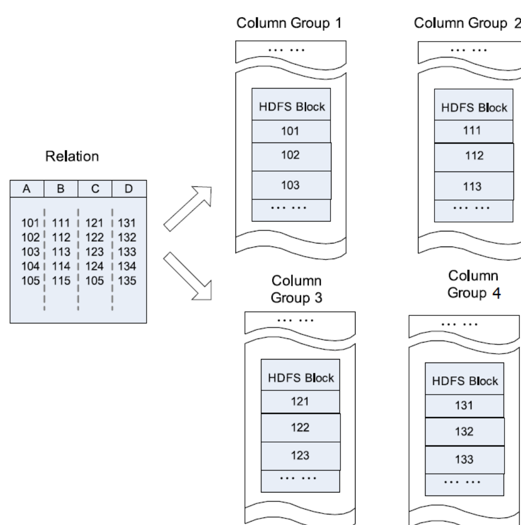

Figure 2.3: Column-oriented data layout

- **Column groups**: This layout consists in organizing all columns of a dataset in different groups of columns. Different column groups may have overlapping columns (see Figure 2.4). The data inside each column group is not tied to any specific data layout: it may be stored in a row or column-oriented way. An advantage of column groups is that it can avoid the overhead of record reconstruction if a query requires an existing combination of columns. However, column overlapping in different column groups may lead to redundant space utilization. Although supporting a better compression than a row-oriented layout, the compression ratio achieved with column-groups is not as good as with a pure column-oriented layout since there is a mix of data types.

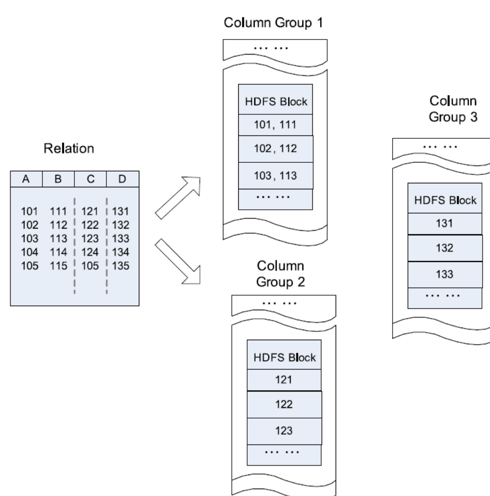

Figure 2.4: Column groups data layout (He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011)

- **PAX** (Ailamaki, DeWitt, Hill, & Skounakis 2001): This hybrid layout combines the ad-

vantages of both row-oriented and column-oriented layouts as show in Figure 2.5. All fields of a record are on the same block as the row-oriented layout, providing a low cost of record reconstruction. However, within each disk page containing the block, PAX uses mini-pages to group all values belonging to each column. This enables to read only the mini-pages that contain the columns needed for the query at hand.

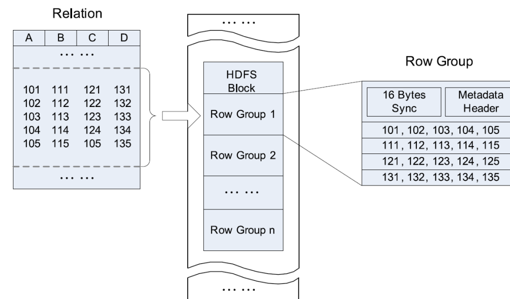

Figure 2.5: PAX data layout (He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011)

2.5.3 Co-location

The data placement policy of Hadoop was designed for applications that access one single file (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010). Furthermore, HDFS distributes the input data among the storage nodes in an oblivious way, i.e., it does not look to the content of the file to decide how to store it.

Co-location is a technique that attempts to ensure that related data is stored together on the same nodes. As a result, applications that require the combined processing of this related data can execute without needing to migrate data during the execution of the job.

2.6 Relevant Systems

In this section, we present some of the most relevant works that use the aforementioned techniques to optimizing the Map phase of selective workloads in MapReduce.

2.6.1 Hadoop++

Hadoop++(Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010) is a system designed to improve Hadoop performance. The goal of its authors is to achieve with a performance

comparable to that of a parallel database management system (DBMS). The authors propose the Trojan Index whose goal is to integrate record-level indexing capability into Hadoop, such that only the records that are relevant for a given job are accessed.

This capability is added to the Hadoop framework without modifying the base code. Hadoop++ adds new fields to the internal layout of splits and it injects appropriate UDFs in the Hadoop execution plan. Since data is stored in a row-fashion way, no data compression is used.

A Trojan Index, as shown in Figure 2.6, is a record-level index over a single attribute that is injected into each split. A Trojan Index is composed by an index data (SData T) plus additional information, such as an header (H) containing the indexed data size, the index size, the higher and the lower indexed attribute values, the number of records, and a footer (F) that identifies the split boundary.

To execute a given job, the footer (F) is used to retrieve all the indexed splits over which the Map tasks are going to be created. When processing a split, the header (H) is used to compare the lower and the higher attribute values existing in that split with the low and the high selection keys of the job. If the two ranges overlap i.e. if the split contains relevant records for that job, the Trojan Index provides the offset of the first relevant record. The next records inside that split are then sequentially read until finding one with an attribute value higher than the high key of the job.

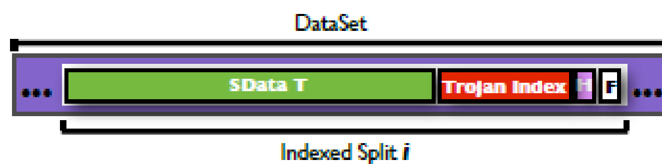

Figure 2.6: Indexed Data Layout (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010)

The creation of a Trojan Index is done at load time through a MapReduce job with proper UDFs: The Map phase takes one record and emits a pair $\langle$splitID+$\mathcal{A}$, record$\rangle$ as the intermediate key-value pairs. The key is a concatenation of the attribute, $\mathcal{A}$, over which we are building the index, and the identifier of the split, splitID, whereas the value record contains all the attributes of that record. For each Reducer to process roughly the same number of splits, the hash re-partitions the records among them according to the splitID. In the Reduce phase, so each

Reducer receives the records sorted by the attribute $\mathcal{A}$, another proper UDF is used. In order to recreate the original splits as loaded into the system, a User-Defined Group function aggregates the records based on the splitID. It is worth noting that now those splits are internally sorted. Finally, each Reducer applies an index builder function over each split which builds the Trojan Index. So this new indexed data could be used in further queries, the Reducers' output is stored on the distributed file system.

Discussion The Trojan Index is an index that allows to sequentially read the relevant records within each split. By storing data in rows, no record reconstruction is needed. With a record-level index, Trojan Index does not avoid passing through all the splits. Also, as a row-store, all the attributes have to be read from disk. However, this improvement is based on the assumption that both the schema and the MapReduce jobs may be known in advance. Also, the solution suffers from being static: the Trojan Index does not take into account the current workload to somehow change the attribute index.

2.6.2 LIAH

LIAH (Lazy Indexing and Adaptivity in Hadoop) (Richter, Quiané-Ruiz, Schuh, & Dittrich 2012) is a system that augments Hadoop with indexing capability. It makes use of a lazy indexing approach, to avoid incurring the overhead of creating the index at load time. Furthermore, LIAH can adapt dynamically to the workload, avoiding the need to know beforehand the access patterns to the data, as opposed to systems such as Hadoop++ (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010).

In LIAH, the indexes are created on a record-level and different attributes may be indexed leading to the existence of multiple indexes. LIAH takes advantage of the fact that data in MapReduce is replicated, to create different indexes for each replica of a block.

The index creation requires minimal changes in the MapReduce pipeline in order to build indexes as byproduct of job executions, as explained below:

First, in order to adapt to the workload, LIAH interprets the attribute accessed on a selection predicate as a hint for what may be an index that would speed up future jobs: For each block, if there is a suitable index for that attribute, the Map task is created on the node where the data block with that index resides. Otherwise, a Map task is scheduled on one of the nodes storing

that block without an associated index. That Map task will then fetch its block from disk to main memory, as normally, and feeds each record to the Map function. The most important feature of LIAH is that, after an unindexed block has been processed by the Map function, it is fed to the components that are responsible for creating the missed index on that block. In the end, the block becomes also indexed with no additional I/O operations. The NameNode is then informed about this newly indexed block so that future jobs may take advantage of it.

Second, an entire dataset is not indexed in a single step. Instead, LIAH creates the indexes in an incremental manner across multiple job executions, in order to avoid delaying the performance of the first jobs. For this, it uses an *offer rate*, ρ, meaning that it creates the index on one block out of ρ blocks in each MapReduce job.

In what concerns the data layout in LIAH, each row is first parsed according to the user-specified schema, followed by a conversion to PAX representation that avoids unnecessary column reads. Even though PAX allows for a good column-wise data compression (Ailamaki, DeWitt, Hill, & Skounakis 2001), compression techniques were not addressed in this work.

Discussion In contrast to previous works (Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012; Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010) that build static indexes, and require to know the profile of the MapReduce jobs in advance, LIAH can adapt to changes in workloads. Also, by piggybacking indexing on the I/O operations of Map tasks, the index creation of LIAH imposes little overhead. The PAX representation used by LIAH, allows to read only the required attributes. However, as the indexes are within blocks, blocks always need to be read, even if they contain no data required by the job. As opposed to previous works, LIAH requires changes on Hadoop, in particular, on the processing pipeline of Hadoop. Therefore, the code of LIAH needs to be kept up-to-date along with Hadoop.

2.6.3 E3

The E3 (Eagle-Eyed Elephant) framework (Eltabakh, Özcan, Sismanis, Haas, Pirahesh, & Vondrak 2013) uses split-level indexing techniques, whose goal is to access only the relevant data splits for the given queries.

In order for the E3 framework to eliminate as much as possible the need to process irrelevant splits, it builds indexes over all attributes in the dataset. These multiple indexes are then

complemented with novel techniques involving domain segmentation, materialized views, and adaptive caching:

- **Range indexing for numerical attributes:** A range index consists in multiple ranges for each numerical attribute in each data split. These ranges are generated by a one-dimension domain segmentation technique that given a set of values $\{v_1,...,v_n\}$ and a maximum number of ranges, k, it returns at most k ranges such that they are "as tightly as possible" containing $\{v_1,...,v_n\}$. (see Figure 2.7). This index allows to eliminate splits containing no ranges hit by any equality or range predicates.

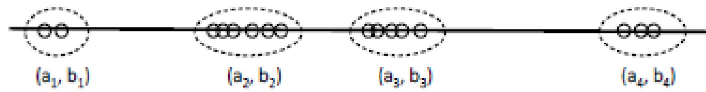

Figure 2.7: Creating range index (Eltabakh, Özcan, Sismanis, Haas, Pirahesh, & Vondrak 2013)

- **Inverted indexes for string attributes:** An inverted index is built for each string attribute. In this case, an inverted index consists in a mapping between a string atom (in this work, an attribute value is called an atom) and the splits containing an occurrence of that atom. This inverted index is physically represented by a fixed-length bitmap with so many bits as the number of splits in the dataset. The bitmap of the atom v has the ith bit set to 1 if the ith data split contains v. This index is appropriated for equality predicates over string attributes.

- **Materialized views for nasty atoms:** A nasty atom v is an atom that appears only a few times in each split but does it so in all of the $\mathcal{N}$ splits of a file. If inverted indexes were used, all $\mathcal{N}$ splits would be accessed. A materialized view is used to overcome this issue: All records containing v are stored in the materialized view. Then, in order to answer queries referring v, only the $\mathcal{M}$ splits composing the materialized view are accessed instead of the $\mathcal{N}$ splits of the original data, where $\mathcal{M} \ll \mathcal{N}$.

- **Adaptive caching for nasty atom pairs:** A nasty atom pair v,w consists in a pair of atoms that individually appear in most of the splits but as a pair, i.e. the atoms jointly, appear in very few records. To handle this, the query workload is monitored and these pairs are cached along with the splits that contain them. Adaptive caching differ from

materialized view since it caches atoms that individually might be spread over almost all the data splits and hence not appear in the materialized view.

All these techniques are used at query time by an algorithm that runs inside an UDF InputFormat and that takes as input a conjunctive predicate in the form of $\mathcal{P} = p_1 \wedge p_2 \wedge p_3 \wedge ... \wedge p_n$, applies the appropriate techniques described above for each individual predicate, and returns a list of the minimal set of splits that must be processed to evaluate the predicate $\mathcal{P}$. Only after this, the JobTracker creates the Map tasks on the splits selected by the algorithm.

In order to build the indexes and the materialized view, the E3 framework requires two passes over the data: First, a MapReduce job is performed to build the inverted indexes and to identify the subset of nasty atoms to be included in the materialized view according to a greedy algorithm. Second, the materialized views is built in a Map-only job containing every record with at least one of the selected nasty atoms.

The design of E3 is not tied to any specific data layout: it uses Jaql for translating the storage format into JSON views of the data what allows it to operate over a vast storage formats. This JSON views consist on arrays of records that are collections of attributes.

Discussion E3 takes into account all attributes of a dataset to build effective indexes and to prevent the processing of irrelevant splits, thus avoiding unnecessary I/O and the cost of launching unnecessary Map tasks. In addition, no record reconstruction is need since records are kept entirely which on the other hand implies that all the attributes have to be read from disk. Furthermore, it does not require any prior knowledge about the query workload. However, this is achieved at the expenses of a relatively costly pre-processing phase, with two passes over the entire dataset. In addition, storing the materialized view, the inverted index, and the range indexes, may lead to a significant storage overhead. Through an UDF that extracts the needed splits and MapReduce jobs to construct the various indexes and the materialized view, we may conclude that E3 was implemented without changing the core of the Hadoop.

2.6.4 Full-Text Indexing

(Lin, Ryaboy, & Weil 2011) proposes an index aimed at optimizing selection-based jobs on text fields to access only the blocks containing text that match a certain query pattern. This

work shows how such index can interact with block compression in order to improve the I/O costs.

This index consists in a single full-text inverted index that works on block-level: given a piece of text the job is interested in, it returns the data blocks containing records where that piece of text occurs. However, in order to obtain compact indexes, the text field in each record is first pre-processed to retain only one occurrence of each term. Consequently, this index can only be used to perform coarse-grained boolean queries.

It is worth noting how the data layout is addressed before we get into how the index works, since the latter depends on the former. The records are serialized by either Protocol Buffers or Thrift serialization frameworks. Data blocks composed by the encoded records in a row-oriented way are compressed using LZO compression. Files in LZO compression format are divided into smaller blocks (typically 256KB) that are automatically handled by an appropriate UDF (InputFormat, more exactly) to be normally processed by Map tasks of Hadoop. Consequently, a split will comprise multiple LZO-compressed blocks.

The inverted index is used, when processing a split, to retrieve the blocks matching the selection criteria. At the end, only the relevant blocks within it are read from disk, decompressed and scanned. To complete the selection, in each relevant block, the encoded records are extracted, decoded and the same selection criteria is applied on each one.

The index creation procedure must be explicitly set up by an administrator who defines which serialization framework to use, how the encoded fields are extracted, among other directives.

Discussion This work addresses an inverted full-index that allows to avoid unnecessary disk I/O readings for blocks containing no interested data for a query at hand. Specifically, the overhead of extract, decode and scan the records inside an irrelevant block is intrinsically avoided as well. Unfortunately, the index creation is not a transparent process as data is ingested. Instead, it must be explicitly set up by an administrator.

In the Hadoop framework, each Mapper unions all the blocks that comprises its split and extracts the records from it.

We believe that through a proper UDF, a Mapper is able to skip ahead the irrelevant blocks when processing a split, so it was not necessary to modify Hadoop.

2.6.5 Trojan Data Layouts

(Jindal, Quiané-Ruiz, & Dittrich 2011) proposes the usage of a new data layout for improving selection-based jobs. This layout consists in organizing the data inside each HDFS block in column groups, allowing to access only the relevant columns to answer a query at hand. This data layout, named Trojan Layout, is motivated by the observation that MapReduce wastes a significant amount of time reading unnecessary column data, due to the records being stored in a row-oriented way.

Externally, Trojan Layout follows the PAX philosophy in the sense that it keeps the same data in each HDFS block as the row-oriented layout. However, it exploits the fact that each block in HDFS is typically replicated three times for fault-tolerance proposes: each replica is internally organized in different column groups, each one aimed to improve the performance of a different type of queries. As we will see later, this system was designed be aware of the schema and queries access patterns. Despite data being stored in a column-fashion way, enabling to achieve a good compression ratio, compression techniques were not addressed by the authors.

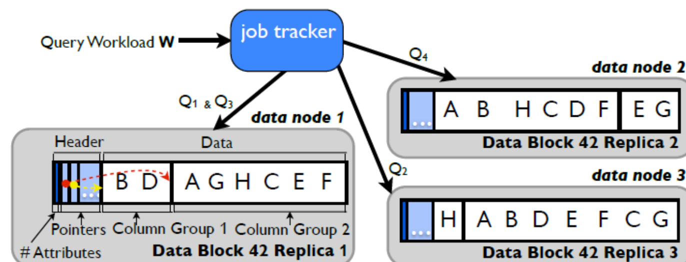

Figure 2.8: Per-replica Trojan Layout (Jindal, Quiané-Ruiz, & Dittrich 2011)

As Figure 2.8 depicts, the Trojan Layout can be divided in two main components: an header and a set of column groups. The header contains the number of attributes stored in a data block and a several attribute pointers, each one pointing to the column group that contains that attribute. Each column group is internally organized in a row-oriented way.

At query time, the attributes needed to be read are identified. Then, the best Trojan Layout is picked according to those attributes. Finally, the Map tasks are scheduled to the nodes containing the data block replicas containing the selected Trojan Layout. At the end, through a properly designed UDF, only the relevant column group is brought to main memory.

The creation of this layout requires some pre-processing in order to identify the best query

	A	B	C	D	E	F	G	H
Q1	1	0	1	0	1	1	1	1
Q2	1	1	1	1	1	1	1	0
Q3	1	0	1	0	1	1	1	1
Q3	0	0	0	0	1	0	1	0

Table 2.1: Example of access pattern. A given cell has a value 1 if a query accesses that attribute, otherwise it has value 0

groups and reformat the data blocks to fit those query groups. Users have first to provide the query workload, the schema of their datasets, and the replication factor to a design wizard called Trojan Layout Wizard (TLW). Then, TLW groups the queries based on a query grouping algorithm that takes into account their access pattern. Finally, the column groups are created in order to speed up the generated query groups. For instance, in Table 2.1, attributes A, C, E, F, G and H, are co-accessed in queries *Q1* and *Q3*. Thus, these attributes are grouped to speed up queries *Q1* and *Q3* as Figure 2.8 shows.

Discussion In this work, data is organized according to the incoming workload. Therefore, for a given query, the most suitable column group is chosen, avoiding to read unnecessary column data. This is achieved without changing the Hadoop processing pipeline but requires a Trojan Layout aware HDFS. On the other hand, record reconstruction of conditional queries may not exhibit good performance: if a query needs to access a column group only when a given record meets some condition, that column group has to be brought from disk and reading its data value by value until reaching the correspondent values of the record that satisfied the condition. This approach does not provide any mechanism to avoid passing through irrelevant HDFS blocks which would speed up the execution time of selective jobs. The major disadvantage of this system is that it assumes that the query workload can be known in advance and remains immutable along time leading to static column groups, an assumption that may prove too strong for some use cases.

2.6.6 RCFile

The RCFile (Record Columnar File) (He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011) is a system that introduces a novel data placement structure implemented on top of HDFS designed to improve selection-based jobs. The data layout introduced in RCFile follows the logic of the PAX format to improve the performance of access to compressed data by allowing to access only

the necessary columns for a given query.

The data layout in RCFile follows the PAX idea of “first horizontally-partition, then vertically-partition”, combining the advantages of both row-store and column-store: As a row-store, it guarantees that data of the same row is located in the same node, requiring a low cost of record reconstruction; As a column-store, it allows to apply column-wise data compression and avoid unnecessary columns reads.

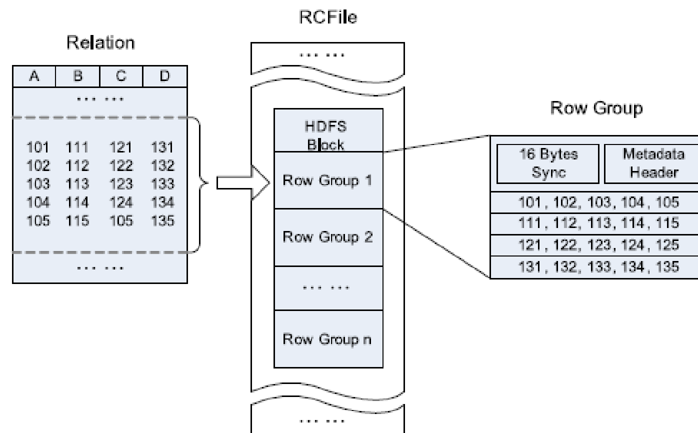

Figure 2.9: Data layout of RCFile (He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011)

As depicted in Figure 2.9, a HDFS block is first partitioned into row groups. Each row group contains a sync marker, a metadata header and table data organized in columns. The sync marker is used to separate continuous row groups whereas the metadata header stores information about that row group such as how many records are in that row group and how many bytes are in each column. The data layout of the table data in each row group is stored in a column-oriented way: All the values of the same column are stored contiguously. Such columns are independently compressed, achieving a good compression ratio.

Data reads for answering a query at hand proceeds as follows: a Map task that is assigned a given HDFS block sequentially reads each row group inside it. To process a row group, through its header, only the columns that are going to be useful are loaded and decompressed.

To convert the raw data into this data format when loading a dataset, the only operation RCFile has to perform is re-organize the data inside each row-group.

Discussion RCFile proposes a new data placement structure that exploits column-wise data compression and skips unnecessary column reads and also avoids the network costs of record reconstruction as a row-store. RCFile is implemented through UDFs requiring no changes on

the processing pipeline of Hadoop. Although a good record reconstruction cost is achieved by avoiding to do it through the network, in RCFile, as Trojan Data Layouts, whenever a record reconstruction is needed, it has to fetch the required columns and going through all their data until the values of the record that triggered the reconstruction have been read. Also, even the HDFS blocks containing no useful data have to be scanned. Its largest disadvantage is that since extra metadata such as the header and sync marker need to be rewritten for each row-group, it leads to additional space overhead.

2.6.7 CIF Storage Format

CIF (Floratou, Patel, Shekita, & Tata 2011) is a column-oriented storage format for HDFS that supports co-location through a customized placement policy, data compression, and optimized readings. By gathering this multiple components, CIF provides efficient access to the relevant column data, improving the Hadoop responsiveness when executing selection-based jobs.

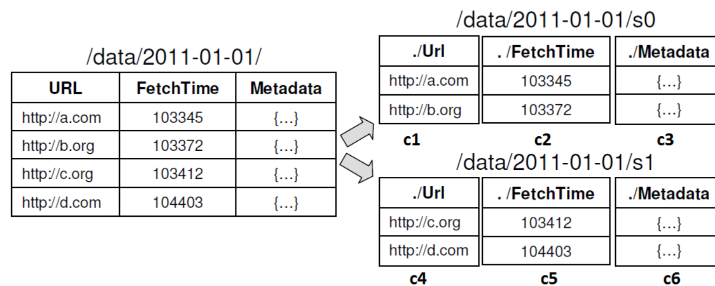

Figure 2.10: Data layout of CIF (Floratou, Patel, Shekita, & Tata 2011)

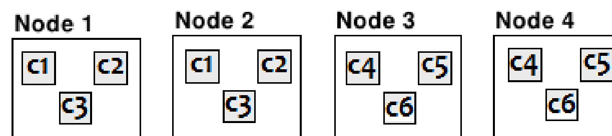

Figure 2.11: Co-location with CIF (Floratou, Patel, Shekita, & Tata 2011)

The idea of CIF is that a dataset is first horizontally partitioned into splits, as shown in Figure 2.10, and each data split is stored in a separate sub-directory. Each column of a given

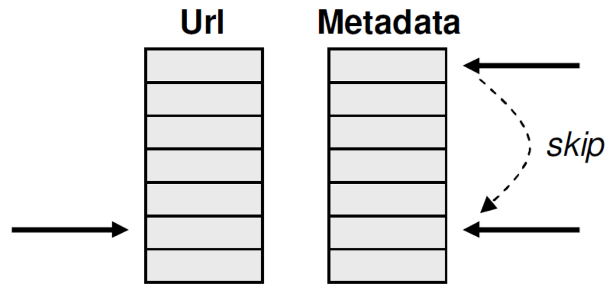

Figure 2.12: Skip list (Floratou, Patel, Shekita, & Tata 2011)

split is individually stored as a single file within the sub-directory associated to that split. On each sub-directory there is also an additional file describing this schema. A new HDFS block placement policy were implemented to guarantee that column files associated to the same split are co-located on the same node to avoid a high record reconstruction cost (see Figure 2.11).

To process a split when a job is submitted, only the column files associated to that split that are needed to produce the result are scanned and the records are reconstructed only with those attributes. When scanning a column file, it uses a lazy record reconstruction technique that applies both a skip list and a lazy decompression scheme. The skip list resides within each column file and enables to skip a set of column values that are known not be relevant. Consider, for example, a query that returns the *Metadata* values of the records whose *Url* satisfy some given pattern. All the *Url* values are read one by one and, whenever a given *Url* value matches the pattern, the skip list enables to jump to the *Metadata* value that corresponds to the same record and that belongs to the file corresponding to *Metadata* column. As a column file may occupy more than one HDFS block, the lazy decompression scheme decompresses a HDFS block only when it is actually going to be read.

When loading data, the application has to pay the price of organizing it in the column-oriented fashion as previously described.

Discussion CIF introduces a new column-oriented storage format for Hadoop that reads only the columns whose attributes are referred in a job configuration. Moreover, it combines a lazy record construction that allows to eliminate disk I/O and mechanisms to co-locate column data that enables reconstructing records without network transfers. Even tough not being as good as a row-store, CIF employs a skip list to efficiently skip irrelevant column values when reconstruction records which implies accessing to different files. In addition, when reading a

column, there is no way of avoiding to pass through the irrelevant HDFS blocks that comprise that column. These techniques have been added without modifications to the core of the Hadoop.

2.6.8 Comparative Analysis

Most of the related work is aimed at avoiding the need to perform a full scan of the input data in the cases where only a fraction of the input is relevant for the desired output. The main approach that has been pursued to achieve this goal consists in building indexes that allow to identify where the needed input is located.

Indexing does not change the data layout. Therefore, even if it allows to retrieve only the records with relevant input, it still requires the application to read undesired columns in the target records. By changing the data layout and storing the input "by-column", it becomes possible to read only the columns that are relevant for the computation. Furthermore, since data in a column belongs to the same domain, this layout has the potential for offering better compression efficiency.

Data co-location has also proved to be an useful technique to optimize the storage of data using a column-oriented layout (by storing columns of the same set of records on the same nodes).

One important aspect of most of the systems we have surveyed is that they require prior knowledge of the schema definition and of the query workload. Without a predefined schema, the indexing, data layout, and co-location optimization mechanisms cannot be applied.

In addition, with the exception of LIAH (which works online and incrementally), these optimization techniques require the execution of a pre-processing phase, where indexes are built, the data layout is changed, or the data is moved to ensure proper co-location. In Hadoop++, for instance, the input data is first transferred through the network and rewritten again into the HDFS which leads to a significant loading cost. Still, according to (Pavlo, Paulson, Rasin, Abadi, DeWitt, Madden, & Stonebraker 2009), most applications are willing to pay a one-time loading cost to reorganize the data if the dataset are going to be analysed multiple times. It is also worth noting that the adaptive approach of LIAH could be integrated into other approaches such Trojan Data Layouts, to interpret the access patterns and adapt the existing column groups to better serve the workload of future and similar queries.

Table 2.2: State of the art systems

		Hadoop++	LIAH	E3	Full-Text Indexing	Trojan Data Layouts	RCFile	CIF
Approach		Access only the relevant records	Access only the relevant records	Access only the relevant splits	Access only the relevant blocks	Access only the relevant columns	Access only the relevant columns	Access only the relevant columns
Use Cases		Selection	Selection	Selection	Selection	Selection	Selection	Selection
Techniques								
Indexing	**Yes/No**	Yes	Yes	Yes	Yes	No	No	No
	Index-level	Record	Record	Split	Block	-	-	-
	Multiple/Single	Single	Multiple	Multiple	Single	-	-	-
Data Layout	**Data Organization**	Row	PAX	Row	Row	Column groups	PAX	Column
	Data Compression	No	No	No	Yes	No	Yes	Yes
Co-location		-	-	-	-	-	-	Based on split-directory
Overall								
Changes Hadoop		No	Yes	No	No	Yes	No	No
Previous Knowledge		Schema and Workload	Schema	Schema	Schema	Schema and workload	Schema	Schema
Pre-Processing		Yes	No	Yes	Yes	Yes	Yes	Yes
Static/Online		Static	Online	Online (cache)	N/A	Static	N/A	N/A
Record Reconstruction		-	Within HDFS block	-	N/A	Within HDFS block	Within HDFS block	From different files but with a skip list
Overheads		Reads irrelevant splits and attributes	Reads irrelevant HDFS blocks	Storage; Reads irrelevant attributes	Extensive manual configuration	Reads irrelevant HDFS blocks	Storage; Reads irrelevant HDFS blocks	Reads irrelevant HDFS blocks

Despite of Hadoop having a strict processing pipeline with a structure hard to modify, UDFs allows to inject arbitrary code into this framework turning it into a more flexible distributed runtime. In fact, these UDFs may suffice for some systems we presented such Hadoop++ or CIF, but not for others that needed to tune other aspects of Hadoop. In the case of LIAH, the changes on the framework were needed to index the data online according to the current workload and without any extra I/O operation. Trojan Data Layouts on the other hand had to add extra metadata information to the HDFS.

Summary

Due to the increasing importance of being able to process big data, MapReduce has become widely adopted, mostly due to its scalability and ease of use. In this chapter we introduced MapReduce and surveyed systems aimed at overcoming limitations of the basic MapReduce model in what regards the access to data. Examples of techniques that have been incorporated in MapReduce include implementing indexes, optimizing the data layout, or co-locating related data on the same location.

The next chapter will introduce the architecture and implementation details of our system.

3 ShortMap

3.1 Introduction

An interesting approach that, to the best of our knowledge, has not been explored is combining per-node indexes with a column-oriented data layout: indexes do the filtering on the data to be scanned and a column-wise layout allows to achieve good data compression and to read only the needed column data, reducing even more the I/O costs.

We now present ShortMap, a system that embodies a complementary and coherent set of techniques that are aimed at improving the Map phase of jobs that only manipulate a fraction of the dataset. ShortMap combines the following mechanisms:

- *Data Layout:* ShortMap organizes the dataset in a way that promotes locality. We achieve this by storing table contents by columns instead of by rows.
- *Data Grouping:* ShortMap groups similar data at each node, to improve the effectiveness of the indexing mechanism without compromising load-balancing.
- *Indexing:* ShortMap creates local indexes, from the data that is stored at each node. Indexes are created and maintained for the most relevant attributes.

We discuss the rationale, and the details, of each of these mechanisms in the following subsections.

3.2 Data Layout

In a MapReduce system, the input data can be modelled as a set of tables, where each table is composed of multiple columns, or attributes. One of these attributes, typically the first, is named the key and identifies each record, or row, of the table. Tables are typically very large

and must be stored in multiple data blocks. There are mainly two approaches to map the table content into blocks: row-oriented or column-oriented.

In the row-oriented data layout all attributes of one record are stored sequentially, and multiple records are placed contiguously into disk. Row-based systems are designed to efficiently process queries where many attributes of a record need to be processed at the same time, since an entire record can be retrieved with a single access to disk.

On the other hand, it has been shown(Floratou, Patel, Shekita, & Tata 2011), that a column-oriented layout is particularly well suited for selection-based jobs that access a small number of columns, since columns that are not relevant for the intended output are not loaded from disk and filtered through by the job, reducing the execution time of a MapReduce job. Furthermore, this layout achieves a higher compression ratio since the data domain is the same in each column, hence each column tends to exhibit low information entropy(Abadi, Madden, & Hachem 2008). The better the data is compressed, the shorter the I/O time spent in reading it from disk (reading data from disk is I/O bound thus can be balanced with CPU).

Unfortunately, since each column in a dataset may represent data with different lengths, a naive partition of the dataset in columns may cause column blocks to become unaligned, complicating the process of building whole records from partitioned datasets. In ShortMap, we avoid this drawback by first partitioning the dataset horizontally, by creating *row groups*, and only then each row group is vertically partitioned by columns, each one being stored in a different file (Figure 3.1). In this way an attribute file is only read when a given query refers to the corresponding attribute, skipping data belonging to the other attributes that are irrelevant for the query. Since by default HDFS places blocks in a random way across all data nodes, ShortMap also includes a row-group aware Block Placement Policy, to make sure that blocks corresponding to the same row group are placed in the same node. This allows our system to use a columnar layout without having to fetch data from other nodes when a full record is required by the job and must be reconstructed from several columns.

3.3 Data Grouping

The grouping component of ShortMap is responsible for rewriting the input data into a format which favours the usage of column-oriented indexes. In fact, columnar indexes may be

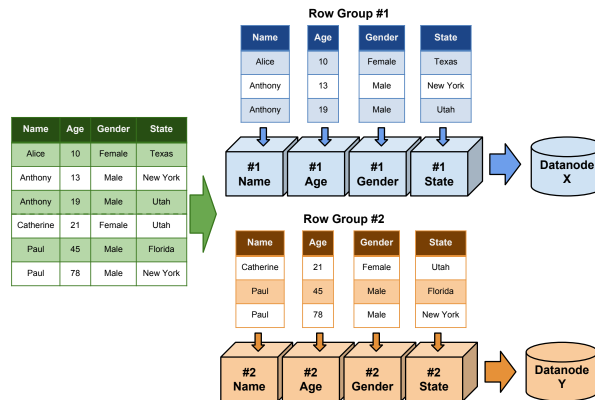

Figure 3.1: Row groups in ShortMap.

quite ineffective without the use of some kind of grouping: On very large datasets, the frequency of most values is significantly larger than the number of row groups the data is partitioned into, which causes any value to be an index "hit" for most row groups.

To make sure that any value in the dataset will be present in the smallest number of blocks as possible, ShortMap reads the whole dataset, identifies records which share the same value for the indexed attributes, and outputs them in a new order such that records with the same value will be contiguous in the dataset. This re-ordering is done *locally*, for each node in the system. The use of local grouping has two significant advantages with regard to an alternative global sorting: not only it makes grouping inexpensive, but also preserves the good load balancing achieved by HDFS's initial distribution of data.

3.4 Indexing

ShortMap uses indexing to avoid loading the whole dataset into main memory during the Map phase. This is accomplished by using the index as an indicator if a given block is relevant for the job being processed.

ShortMap indexes are fully decentralised in the sense that each node builds its own index based on the data it stores. This design decision simplifies the creation of indexes, and makes for quick local index lookups. In terms of structure, the indexes are actually inverted indexes, where each entry maps an attribute value to its location in the dataset. In fact, there are two possible representations for this pair, each with different tradeoffs. The first is to map the attribute value to a row group identifier. This identifier represents the first row group where the

value occurs. Since data in each node in ShortMap is grouped by attribute value, the system is guaranteed to scan through all records which have a given attribute if it starts scanning at the row group pointed by the index and stops as it reaches a different value. The second possible representation is to use a pair of ⟨ *RowGroupID, [column1-offset, column2-offset, . . . , columnN-offset]* ⟩ attribute value. The first element of the pair points to the first row group which contains the attribute value. The second element is a list with length equal to the number of columns in the dataset, such that each entry corresponds the first offset containing the value within the block of the corresponding column. Even though the second representation has the potential to create larger indexes (since it must contain an entry for each column in the dataset), it also has the potential to achieve better performance as it allows ShortMap to directly retrieve a record from a specific offset in a block (without having to filter through all records which may appear before the target in the block).

In order to save memory during query time, the indexes are stored on persistent storage along with the data. The indexes are stored partitioned, such that a small set of pairs is kept on each file. Thus, when a task is scheduled at a node, only the index pairs corresponding to the attribute value queried for by the Map task are brought to memory from disk, to determine if the blocks associated with the split of the Map task should be read. In order to achieve efficient index querying, ShortMap keeps an in-memory cache of the most recent attribute values queried for using a least recently used policy. This design simplifies the management of indexes, since it allows a node to keep the index pairs in memory during the whole job, without requiring information on job completion (which in MapReduce is only available at the master, the JobTracker).

3.5 Other ShortMap Aspects

In the previous section we have described the main mechanism that we have incorporated in ShortMap. In this section we discuss additional issues, such as the use of replication, the storage of indexes, the validation of blocks before transfer, and the pre-processing step required by ShortMap.

3.5.1 Replication

Hadoop supports replication, allowing each block to be replicated in (a configurable number of) R nodes, such that when a node fails the data may be recovered from other nodes in the system. This mechanism is also leveraged when scheduling tasks, since when choosing a node where to spawn a task, the scheduler gives priorities to nodes which already store the data. ShortMap also allows each block to be replicated in R different nodes with similar advantages. However, since each node groups the data it stores, replication in ShortMap must be made on a per-node basis instead on a per-block basis. Furthermore, similarly to other state of the art systems (Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012; Richter, Quiané-Ruiz, Schuh, & Dittrich 2012), ShortMap also allows each replica to group the data according to a different attribute, in order to allow for optimised queries for more than one attribute.

3.5.2 Index Storage

As mentioned previously, ShortMap's indexes are stored on disk and loaded as needed during querying. In order to be able to load portions of the index, index pairs are saved into different files (which act as buckets) depending on a fixed portion the hash value of the key (i.e. the value of the attribute).

3.5.3 Block Transfer Pre-Validation

Regarding index usage, notice that ShortMap does not make use of a centralised index but MapReduce's model supports nodes retrieving data blocks from other nodes in the system. In particular, when a node has finished processing all tasks related with its data, the JobTracker will assign it tasks related with data of nodes which have fallen behind in processing. When processing selective queries, this may cause a block with no relevant information to the worker to be transferred over the network. This is inefficient, specially because we have found the network to be one of the key bottlenecks in the execution of a MapReduce job (this observation concurs with other research in the area (Palanisamy, Singh, Liu, & Jain 2011)).

To avoid this problem, in our implementation we modified Hadoop such that the node that hosts the data makes a local check, by loading the corresponding indexes to determine if the block is relevant or not, before transferring the block (avoiding the transference if it includes no

relevant content to the requesting worker). The results presented later use a version of Hadoop with this optimisation. Note, however, that ShortMap is not tied to this implementation decision.

3.5.4 Pre-Processing

ShortMap requires the execution of a data pre-processing stage to build the indexes and re-format data blocks by splitting the rows into columns and grouping similar data. Since this pre-processing stage does not involve any exchange of information among nodes, it can be performed efficiently, and can actually leverage on MapReduce itself.

In more detail, after the data is loaded to HDFS, we run a pre-processing MapReduce job configured such that each node in the system acts as a mapper as well as reducer. The job goes through all lines in all input files (i.e. all records stored by the node), and in the Map phase outputs a pair $< (\mathit{ID}, \mathit{value}), \mathit{List} < \mathit{field} >>$ associating the record itself, formatted as a list of values for fields, with an intermediate key. The intermediate key makes sure that the record will be processed by the reducer co-located with the mapper, since it contains the identifier of the node storing the data; in order to allow the reducer to perform the grouping, the Reduce key also contains the value of the record for the attribute for which the data will be grouped. Since the attribute to be used for data grouping must be defined at the stage of pre-processing, the user is responsible for selecting which attribute is most relevant for indexing. In our prototype, this attribute is included in a configuration file, but in a working system it should be included as a parameter of the *copyFromLocal* command of HDFS.

At the Reduce phase, each reducer receives several sets of records, grouped by attribute value. Thus, the reducer's task is to output each field of the record to a different HDFS block. Notice that since each column contains different types of data, each output block may grow at a different rate. To guarantee that all blocks will fit in an HDFS block and that the several blocks corresponding to a row group are aligned with each other, as soon as any column's content reaches the size of an HDFS block, a new row group is created. This mechanism follows a design similar to other state of the art column-oriented storage systems (Floratou, Patel, Shekita, & Tata 2011; Jindal, Quiané-Ruiz, & Dittrich 2011; He, Lee, Huai, Shao, Jain, Zhang, & Xu 2011). Throughout this process, ShortMap captures the starting positions of each different value, such that it can populate the block index.

Although it is required to transform all the data in the system in this pre-processing phase, it is worth noting that this cost represents an one-time cost that is paid at load time. This way, we assume that the gains after querying the data multiple times will be higher than this pre-processing cost.

3.6 ShortMap Implementation

ShortMap has been implemented as a set of extensions to the Hadoop framework and the prototype is available for download[1].

ShortMap extensions consist of roughly 6600 lines of code. In this section, we start by giving an overview of some Hadoop entities and modules, and next we describe the most important tweaks done on Hadoop to support ShortMap functionality.

3.6.1 Hadoop Brief Overview

As presented in Section 2.3.1, the Hadoop Distributed File System is composed by a NameNode, a SecondaryNameNode and a set of DataNodes. Besides of storing the metadata of all files in the HDFS, the NameNode is also responsible for placing HDFS blocks when these are loaded into the system which, by default, is done in a random way across all DataNodes. Since our system has more specific data placements, we introduced a new block placement policy to ensure that files belonging to the same row group are placed on the same DataNode, in order to allow for local record reconstruction in cases where more than one attribute is queried.

The processing layer is composed by a JobTracker and a set of TaskTrackers (see Section 2.3.2). The user starts by submitting a job which contains information about the path of the input data, a reference to a custom class (`InputFormat`) responsible for handling them, the Map and Reduce functions and other relevant information. Upon job submission, the JobTracker divides the input data into splits according to the `InputFormat`, which are then processed by the Map tasks that are being assigned to TaskTrackers as these are ready to accept new ones. Although there are already some pre-defined `InputFormat` classes, we had to program a new one to handle the specifics of our data layout. When processing a split, a TaskTracker resorts

[1] The prototype can be downloaded from:`https://github.com/shortmap/shortmap`

to the `RecordReader` that is also instantiated by the `InputFormat`. The RecordReader is used to decompress blocks (if necessary), generate the key/value pairs from the raw input split data and for sending them, one by one, to the Map function. ShortMap requires the `RecordReader` to be able to skip irrelevant input data for a job at hand: both irrelevant row groups and unnecessary attributes. From this point on, ShortMap requires no more changes on Hadoop. The intermediate key/value pairs are then generated by the Map tasks, shuffled and passed to Reduce tasks leading to the final result.

3.6.2 Most Important Tweaks on Hadoop

Regarding the HDFS layer, ShortMap includes a `Row-Group Aware Block Placement Policy` used by the NameNode to make sure that blocks corresponding to the same row group are placed in the same node. No modifications on the core of Hadoop were needed to implement this feature, since custom Block Placement Policies are easily pluggable. Also, in order to evenly distribute all the data among nodes, our block placement policy stores row groups in a round robin manner.

Since ShortMap is based on file manipulation, the most appropriate `InputFormat` is the `FileInputFormat` of Hadoop, that is the base class for all file-based `InputFormats`. However, the `FileInputFormat` goes through all the input files and creates one split per HDFS block. For ShortMap, we defined an `InputFormat` that is row-group aware, in order to create only one split per row group instead of one per HDFS block. Due to the fact that our block placement policy ensures that all the nodes have about the same number of row groups, a given job will generate approximately the same number of splits local to each node. In addition and considering that the data is uniformly distributed among nodes, during job execution, all the nodes will have approximately the same processing and network load. In the most cases, this causes all the nodes finish processing Map tasks at the same time, since the cluster is balanced in terms of data and load.

For ShortMap, we implemented a `RecordReader` with two particular capabilities. Firstly, it performs a local index lookup to check if any row groups assigned to the split contain a relevant entry. In the positive case, the `RecordReader` obtains the offset of the first row to start reading from that point until reaching either the end of the row group or an entry that no longer satisfies the query. Since we allow for nodes to avoid transferring non-relevant blocks when other nodes

are processing the data they store, we have also added a new routine that is called by the processing node so the owner of a given row group performs the index lookup and sends it the relevant portion of the row group if relevant; or a negative answer otherwise. Before performing an index lookup from disk, the owner of a row group checks whether its index cache suffices or not, i.e. if it contains the relevant portions of its stored data for the queried attribute value. If not, the node needs to calculate the hash code of the queried attribute value and fetch from disk the index portion correspondent to that hash. The relevant row groups and their offset for this query are then cached for future tasks and jobs. This way, during the whole execution of a given job, only the first index lookup in each node will trigger a disk reading. The cache will maintain the information about the relevant data at least until the end of the job. Furthermore, next following jobs querying for the same attribute value will have the caches satisfying them. The second capability of the `RecordReader` is that it extracts from the job configuration the attributes referred by the query and passes the values corresponding to those attributes to the Map function.

It is interesting to note that, although these changes can achieve significant improvements of the MapReduce performance (over the standard Hadoop implementation), the majority of them require no modifications on the core of Hadoop but, instead, can be implemented using the UDFs provided by Hadoop.

Summary

In this chapter we have been through the design and implementation of ShortMap. In the first place, we described the three main mechanisms of ShortMap: data layout, data grouping and indexing. Next, we discussed additional issues, such as the use of replication, the storage of indexes, the validation of blocks before transfer, and the pre-processing step required by ShortMap. Then, we described the most important implementation details behind our prototype of ShortMap.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

4.1 Introduction

In this section, we start by presenting results for a mini-benchmark which allowed us to decide the index designs. Next, we show how each part of ShortMap contributes to its overall result by comparing with other state of the art solutions specifically concerned with optimizing the Map phase of selective workloads. We also present results for how our system compares with an unmodified Hadoop version. Finally, we compare ShortMap with a more recent framework called Spark.

All experiments have been performed using a cluster of 20 virtual machines (deployed on 10 physical machines) running Xen, equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Ubuntu Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet.

For all experiments presented in the current section, we used a sample of the Twitter dataset (https://dev.twitter.com/docs/platform objects/tweets), collected between May and September 2012. This dataset is comprised of 325,333,833 tweets, that correspond to 988GB of raw data, which when compressed with gzip, have a total of 161GB. The tweets are stored in JSON format, each containing 23 attributes (such as an identifier, creation date, hashtags, the text message itself, as well as an embedded JSON object with 38 more attributes about the owner of the tweet such as her language, location, identifier, etc). Due to non-disclosure restrictions imposed by Twitter, we are unable to make the dataset public, but a similar dataset can be obtained by querying the Twitter's API.

All values presented are the average of at least 3 executions, and for our configuration, an unmodified version of Hadoop takes between 1 and 3 hours to process a query.

Our workloads capture scenarios where a provider might offer its infra-structure for their clients to perform data analysis based on a portion of the dataset, such as the demographic they

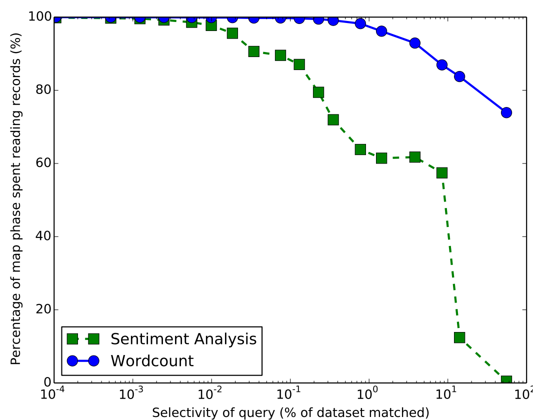

Figure 4.1: Cost of reading data, relative to the duration of Map tasks, depending on job type and query selectivity.

are interested on, a specific location, language or associated hashtag. We use two different types of selective MapReduce jobs; Each type of workload allows to analyse different aspects of the system, according to the amount of processing required by the corresponding Map function. The first workload consists in applying a selective query to the dataset, to retrieve only tweets using a specific language, and then apply a simple word count. This analysis works as a baseline for comparison with a second, more complex and realistic analysis, which consists on calculating the sentiment of users from the corresponding tweets' text. This job mimics the big data processing required to extract scientific results from Twitter datasets (Kloumann, Danforth, Harris, Bliss, & Dodds 2012; Mitchell, Frank, Harris, Dodds, & Danforth 2013; Frank, Mitchell, Dodds, & Danforth 2013; Bertrand, Bialik, Virdee, Gros, & Bar-Yam 2013), and it has significantly higher CPU processing requirements than the wordcount job.

To better illustrate the difference between both types of jobs, Figure 4.1 shows the percentage of the Map phase time that ShortMap spends reading records from disk, when executing the sentiment analysis and the word-counting job. These values, similarly to others in this section, are presented as a function of the selectivity of the query, i.e., to what percentage of the dataset does the value queried by correspond. Since the word-counting job needs less data processing on the Map function, it spends a larger fraction (at least 73%) of the Map phase time reading the necessary records. Lighter jobs are therefore often bounded by the time required to scan the input data. Conversely, sentiment analysis requires more processing on the Map function. This is particularly prominent when processing the text of the users whose language is the most common in the dataset. In this case, since there is a large number of text messages to process,

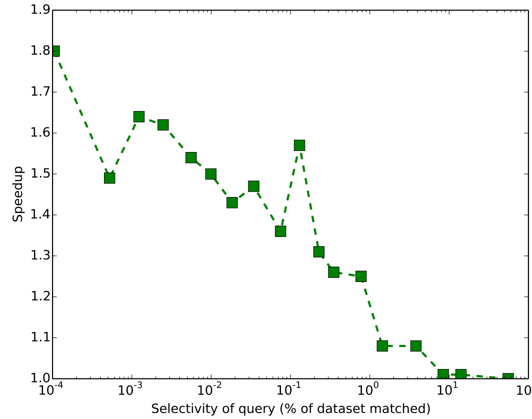

Figure 4.2: Read operation speedup of using a list of offsets over using a simple row group identifier for implementing ShortMap's indexes.

most of the Map phase time is spent processing records and not reading them from disk.

4.2 Index Structures Micro-benchmark

In Section 3.4 we described two possible representations for ShortMap's index. We now present comparative results for the performance of each representation, with the intention of selecting the best option for the remainder of this evaluation.

The two previously presented options consisted of mapping each attribute value to 1) a row group identifier or 2) a row group identifier and a list of offsets. These two options differ in performance and space usage. On one hand, using only a row group identifier means that the size of the index will not depend on the number of attributes in the system but more data must be read at query time. On the other hand, using a list of offsets may allow for performance improvements since the system can seek directly to the relevant records.

In order to better understand the performance tradeoff in these two options, we implemented both mechanisms, executed queries of various degrees of selectivity on ShortMap and recorded the time required to read data from disk. Figure 4.2, presents the read operation's speedup achieved by the list of offsets (option 2) over the row identifier (option 1).

As Figure 4.2 shows, using a list of offsets yields considerable improvements when reading data, especially for queries with high selectivity. This can be explained by the fact that for most scenarios with high selectivity, the option of using only a row group identifier involves always

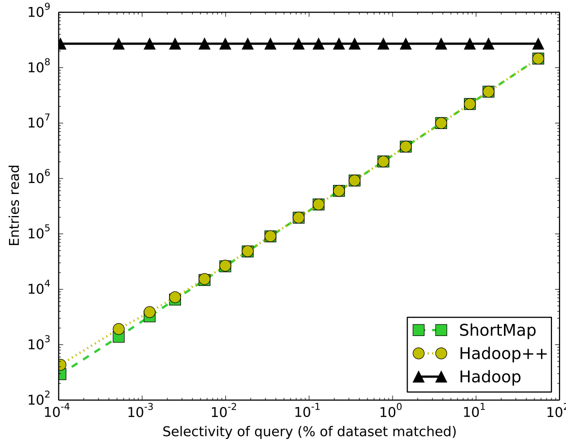

Figure 4.3: Entries read by Hadoop, Hadoop++ and ShortMap.

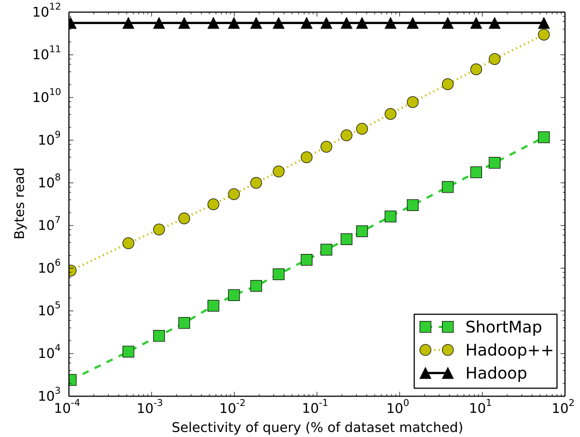

Figure 4.4: Bytes read by Hadoop, Hadoop++ and ShortMap.

reading a whole block, while using an offset allows the system to skip directly to the few relevant records within the block. It is important to notice that since we are using a compressed dataset, the efficiency of the list of offsets index is reduced due to compression limiting the possibility of using the "seek" operation of file systems, but still allows for modest improvements.

Regarding the space usage, we have found that for our dataset, the index built using a list of offsets can be take to 16 times more space. However, this comparison is dependant on the number of dataset attributes, as well as on the size and diversity of the indexed attribute. In Section 4.3.2 we will analyse with more detail these and other tradeoffs. Since the space used by both options takes less than 1% of the size of the dataset, we argue that the space usage loss is shadowed by the performance improvement results, and opt by using the list of offset index for the remainder of this evaluation.

4.3 ShortMap against Indexed Row-Oriented Stores

One of the main concerns of ShortMap is Data-Layout. To achieve better efficiency when reading data, we use columnar storage. In order to evaluate the effect of this design decision, in this section we compare ShortMap with systems which make use of row-oriented layout for the storage and perform lookups using indexes. We study the effect of this decision both in terms of performance as well as space usage.

A prominent and comparable solution to our system is Hadoop++, (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010). Hadoop++ not only uses a row-oriented layout, but also

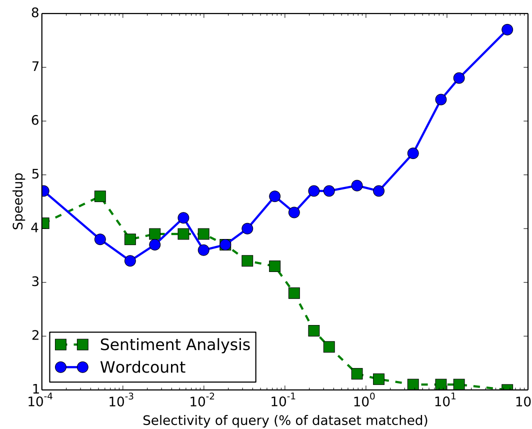

Figure 4.5: Speedup of ShortMap over the Hadoop++.

creates indexes for the items contained in each block. Unlike our solution, Hadoop++ sorts each block by itself, and creates a block-local index, which is appended to the block. In Hadoop++, this index is loaded at query time, and used as a hint to know which parts of the block should be read to answer the query at hand. Since the code for Hadoop++ is not publicly available, we have also implemented a prototype of this system following the specification provided in (Dittrich, Quiané-Ruiz, Jindal, Kargin, Setty, & Schad 2010).

4.3.1 MapReduce Performance

To better illustrate the design differences among ShortMap, Hadoop++, and Hadoop, Figures 4.3 and 4.4 show the number of bytes and entries read by each solution. Hadoop++ reads much less data and entries than Hadoop, and about the same number of entries as ShortMap (since its indexes allow it to bypass non-relevant blocks). Similarly to ShortMap, the number of bytes read is dependent on how common the value queried for is. In terms of read data, the main difference between both systems is that since Hadoop++ uses row storage, it must read all columns of records, whereas ShortMap needs only to read the blocks corresponding to the relevant columns.

Figure 4.5 shows how these decisions reflect in terms of performance of the system. This figure presents the speedup of ShortMap over Hadoop++ while querying for different values. As expected from the analysis of read data, for all executions, ShortMap exhibits speedups over Hadoop++. The performance of both systems tends to be similar when querying for more common items and performing a complex computation (i.e. sentiment analysis) since in that

situation the performance of the system is mostly limited by the processing cost. On the other hand, for the more selective jobs ShortMap shows up to 5 times speedups, due to reading less data as was shown previously.

Interestingly, when for the wordcount job, an unexpected effect comes into play and ShortMap's speedup actually increases as we look at less selective queries. This result is due to the fact that Hadoop++, for the most common attributes, needs to read all the blocks from the dataset. In fact, even not all the records from the blocks are relevant for this query (only 50% of the records are relevant), all the blocks contain relevant records since they are uniformly distributed among blocks. This way, for less selective queries, the performance of Hadoop++ reaches the one of Hadoop, while ShortMap still avoids to read the remainder columns. Overall, for the results presented, ShortMap achieves an average speedup of 4.4 over Hadoop++ for the word-count job, and of 2.4 for the sentiment analysis.

4.3.2 Index Space Usage

In this section we analyse how ShortMap's per-node index compares with a per-block index (such as that used by Hadoop++) in terms of space usage. For each block stored at every node, and for a given indexed attribute, Hadoop++ appends one index to each block after a pre-processing phase where the block is internally sorted. Conversely, ShortMap creates a single index at each node, and the index maps the values to their corresponding row group and list of offsets. As mentioned in Section 4.2, we have also included results for the alternative implementation of our index, which omits the lists of offsets.

Table 4.1 presents the space used by ShortMap's two types of index and those of a per-block index, depending on the attribute used and on the row group size. The two attributes are representative of two types of distributions which are common in our dataset: "User Language" represents a low cardinality attribute, whereas "User Location" represents a high cardinality attribute. The results show that for any attribute and row group size, the best index config-

Table 4.1: Space usage of Per-Node Index and Per-Block Index.

Attribute	User Language			User Location		
Row Group Size (MB)	16	32	64	16	32	64
Per-Node Row Group Index (KB)	4.04×10^{-1}	4.04×10^{-1}	4.04×10^{-1}	5.99×10^{5}	5.99×10^{5}	5.99×10^{5}
Per-Node Offset List Index (KB)	6.47×10^{0}	6.47×10^{0}	6.47×10^{0}	3.83×10^{6}	3.83×10^{6}	3.83×10^{6}
Per-Block Index (KB)	3.61×10^{2}	7.00×10^{2}	1.35×10^{3}	2.87×10^{6}	2.70×10^{6}	2.29×10^{6}

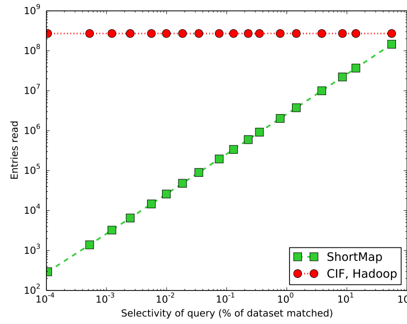

Figure 4.6: Entries read by Hadoop, CIF and ShortMap.

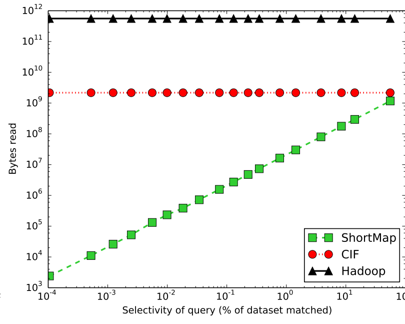

Figure 4.7: Bytes read by Hadoop, CIF and ShortMap.

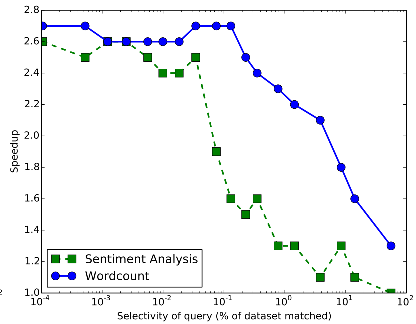

Figure 4.8: Speedup of ShortMap against CIF.

uration in terms of space is to use the per-node index which omits the lists of offsets. This result is due to a) this index having strictly less entries than the block indexes (which may have entries repeated between them); and b) this representation not being dependant on the number of columns in the dataset.

The results also show that the index used in all results of this section are comparable to those of a per-block index for a high cardinality attribute, and considerably better for the low cardinality attribute. It is also important to notice that unlike the per-block indexes, the size of the per-node indexes is independent of the size of row groups. In conclusion, should the space be a high concern in the configuration of the system, a system administrator should opt by using ShortMap configured with row-group-only indexes.

4.4 ShortMap against Column-Oriented Stores

Similarly to ShortMap, other systems have also adopted a columnar storage. In this section we study how our system compares with a state of the art solution which also uses this type of data layout. In order to evaluate the impact of the indexing and data grouping techniques, we opted by comparing our system with CIF (Floratou, Patel, Shekita, & Tata 2011). Since CIF's code is not publicly available, again we have implemented a prototype of CIF using the codebase of ShortMap, by disabling the indexing and data grouping components of our system.

Figures 4.6 and 4.7 allow for a better understanding of how ShortMap compares with Hadoop and CIF in terms of entries and bytes read. Since CIF lacks any kind of indexing, the number of bytes and entries read does not vary as a function of how common the queried value is,

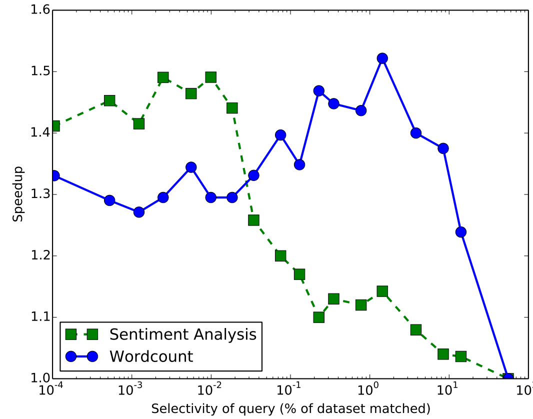

Figure 4.9: Speedup of ShortMap against an indexed column-oriented store.

similarly to unmodified Hadoop. However, the amount of data read by CIF is smaller than that of Hadoop, since it can read only the columns relevant for the query at hand.

Since ShortMap uses indexing and data grouping to reduce the amount of data read, it outperforms CIF for the scenarios where the indexes are more effective. Figure 4.8 depicts the speedup achieved by ShortMap over CIF, for both types of jobs considered: for less selective queries, the speedup is more reduced, while when querying for more uncommon values, the indexes allow ShortMap to achieve a speedup of up to 2.6 over CIF. Similarly to what was observed earlier, for the job with a higher processing cost the average speedup is smaller (about 1.9), while for the lighter job a higher average speedup is achieved (around 2.4).

4.5 ShortMap against Indexed Column-Oriented Stores.

Several state of the art systems opt by using a layout that allows to read only the relevant attributes. Such layout is combined with per-block indexes (Richter, Quiané-Ruiz, Schuh, & Dittrich 2012; Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012). In this section, we study how our system compares with a variant of our implementation of Hadoop++ which instead of storing data per rows, stores it in columnar row groups (similarly to ShortMap), and indexes each row group individually (unlike ShortMap, which groups data and indexes data on a per-node basis). Since the code for neither LIAH (Richter, Quiané-Ruiz, Schuh, & Dittrich 2012) nor HAIL (Dittrich, Quiané-Ruiz, Richter, Schuh, Jindal, & Schad 2012) is publicly available, this prototype represents a simplified version of such systems. The main goal of the study presented in this section is to evaluate the effect of the grouping and per-node index component

of ShortMap.

Figure 4.9 presents the speedup of ShortMap over an indexed column-oriented store. As was observed before, since the less selective queries of the sentiment analysis job are heavily CPU-bound, ShortMap does not present a significant advantage over other solutions. Still, for this job ShortMap can reach up to 50% better performance, and achieves an average of 25% speedup. Two facts contribute to these results. Firstly, using per-block indexes requires the system to load several indexes from disk while processing the job, in contrast with ShortMap which only loads a partial index on the first task associated with the job. Secondly, and more importantly, since the data is not grouped, several blocks may be a match for the query value, which requires the system to open several blocks, to seek to the offset of the value, whereas ShortMap most likely will only load a single block.

For the wordcount job, the costs of not using grouping become more pronounced. Unlike the results achieved for the sentiment analysis, the speedup of ShortMap over an indexed column-oriented store actually increase when the selectivity of the query decreases. These results are strongly tied with how many blocks are matched by the query. For ShortMap, the number of blocks matched is proportional with the selectivity of the query. However, on a system that does not rely on grouping, the number of blocks matched grows superlinearly with the selectivity of the query since the records containing the matched value are distributed randomly across all blocks. This effect is particularly notorious towards the less selective queries, since the number of blocks matched by the query on ShortMap remains very small whereas for the system using per-block indexes all blocks are a match for the queries. This also explains why towards the most common attributes the speedup of ShortMap drops to values close to 1: for this situation, in both systems, a large number of blocks is matched and the cost of reading the data shadows that of seeking inside the blocks. Furthermore, also notice that for the rarest item, the speedup of ShortMap is also more reduced than for the following queries: this is due to this value being present only on a single block for both systems, yielding a similar performance (with a slight edge to ShortMap due to reading only a single index).

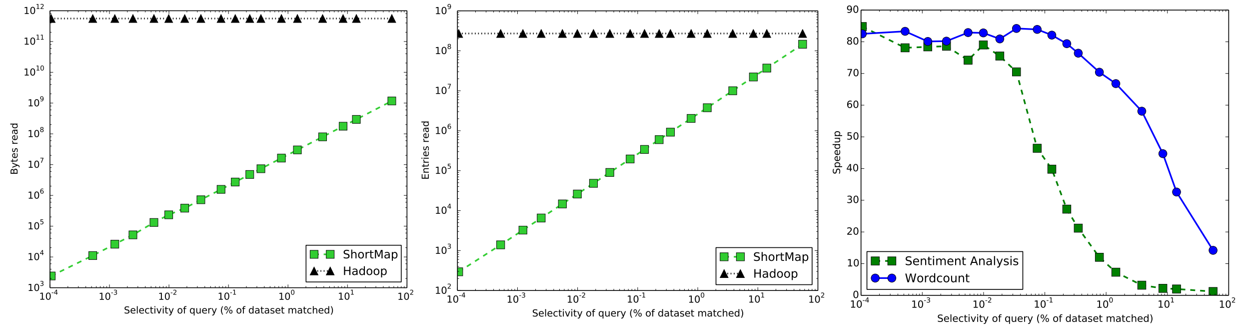

Figure 4.10: Bytes read by Hadoop and ShortMap.

Figure 4.11: Entries read by Hadoop and ShortMap.

Figure 4.12: Speedup of ShortMap over the unmodified Hadoop.

4.6 ShortMap against Hadoop

In this section, we evaluate the overall performance of ShortMap, by comparing executions of our prototype of ShortMap with the unmodified version of Hadoop. We compare both solutions in terms of the amount of data read from disk, as well as the performance for the two query types for different frequencies of values in the dataset.

Unlike Hadoop, ShortMap does not require a full scan over the entire dataset when performing a selective query. As shown in Figure 4.10, this results in ShortMap reading significantly less data than Hadoop. Furthermore, ShortMap is required to filter through less entries, which cuts the CPU cost as well, as shown in Figure 4.11. Notice that even when querying for the most common attribute value, which corresponds to approximately half of the dataset, ShortMap still reads much less data than Hadoop, since it only reads the columns relevant for the query at hand. The impact of this improvement is more notorious in jobs with larger fractions of time spent reading the records, since this is the part of the Map phase time that is shortened.

Figure 4.12 depicts the speedup of ShortMap over Hadoop, depending on the frequency for which the queried value can be found in the dataset. The consequences of ShortMap reading considerably less data than Hadoop can clearly be observed in these results, since ShortMap achieves up to almost two orders of magnitude faster queries. As expected from the analysis of read data, the advantages of ShortMap are less significant when querying for more common values, since the gap between the data read by ShortMap and Hadoop closes.

Still, even when querying for the most common attribute, for the word-count job ShortMap reaches a 14.2 times speedup while reading 460 times less data, 1.79 times less entries,

and processing the same number of entries. In this scenario, given that many entries need to be processed, the processing costs, albeit small, limit the speedup of ShortMap. This fact is particularly evident when looking at the sentiment analysis: ShortMap is only 1.2 times faster than Hadoop for the most common value. Furthermore, notice that due to the higher CPU cost associated with sentiment analysis, the speedup of ShortMap also drops earlier (i.e. for values less common) than for the word-count job. These results lead to the conclusion that even though ShortMap is particularly well suited for selection jobs using uncommon values, the usage of a columnar layout allows it to achieve good speedups even when selecting by the most common values in the dataset.

4.7 ShortMap against Spark

In this section, we evaluate the performance of our system against a different and more recent framework called Spark(Zaharia, Chowdhury, Franklin, Shenker, & Stoica 2010). Apache Spark is an open source computing framework for large-scale data analysis. The main difference between Spark and MapReduce is that Spark was designed for pipeline scenarios. During the first time a job is executed, Spark keeps data in memory through collections of elements called Resilient Distributed Datasets (RDDs)(Zaharia, Chowdhury, Das, Dave, Ma, McCauley, Franklin, Shenker, & Stoica 2012). The aggressive caching allows Spark to avoid reading data from disk in subsequent executions of the same or different jobs that are interested in accessing the same input data or data resultant from previous computations. This means that Spark is more relevant when performing iterative computations over the same data (e.g. machine learning) on a cluster with enough resources to keep the whole dataset in memory. If we consider a single MapReduce job, the key difference between Hadoop and Spark is that the latter holds the intermediate results in memory through RDDs whereas the former writes the intermediate data back on disk. According to its authors, Spark can be 100 times faster than Hadoop in memory and 10 times faster when reading from disk, due to using different policies for fault-tolerance and due to having improved job scheduling mechanisms.

For this evaluation, we used a considerably larger dataset containing 610,955,987 tweets corresponding to approximately 2 TB of raw data. When running Spark, data was also stored using the Hadoop DFS.

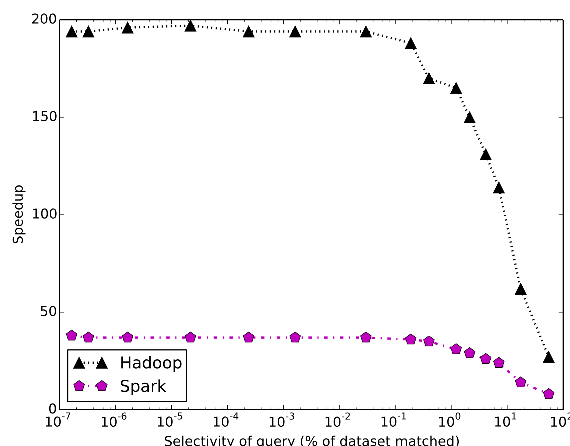

Figure 4.13: Speedup of ShortMap against Spark and Hadoop for the word counting job.

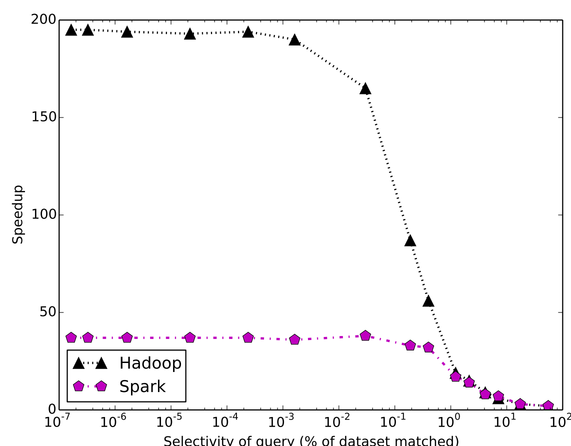

Figure 4.14: Speedup of ShortMap against Spark and Hadoop for the sentiment analysis job.

Figure 4.13 and 4.14 shows the speedup that ShortMap achieves over Hadoop and Spark for the wordcount and sentiment analysis jobs, respectively. As it happened with the smaller dataset, ShortMap is considerably faster than Hadoop. However, for this dataset, the difference is even larger than previously observed mostly because the average content of the tweets is larger, thus increasing the difference between the amount of data that ShortMap and Hadoop read, leading to larger speedups. Even though Spark can achieve better performance than Hadoop due to its improved scheduling and use of in-memory storage of intermediate results, it is still considerably slower than ShortMap. ShortMap achieves speedups from 8 (for the most common value) up to 38 times (for the rarest items), for the wordcount job and between 2 and 37 times for the sentiment analysis job. Analogously to the previous comparisons, the speedup is lower when performing jobs with higher data processing rates. These speedups are achieved because similarly to Hadoop and to some of the state of the art systems previously presented, Spark must read the whole dataset from disk at least one time, in order to perform a selective query.

This shows that even though Spark can considerably outperform Hadoop for most workloads, is not able to perform as efficiently as ShortMap in these selective scenarios, since it must still read all data from disk at least once.

Summary

In this chapter we introduced the experimental evaluation made to ShortMap and its results. First we presented a micro-benchmark in order to select the best index structure for the final prototype of ShortMap. Next, in each section, ShortMap was compared against a different state of the art solution. This allowed us to evaluate individually the effectiveness of each mechanism integrated on ShortMap. In the first experiment we tested the performance of ShortMap against an indexed row-oriented store with the intention of evaluate the effect of using a columnar storage. Next, we compared ShortMap against a column-oriented store so we could test the impact of the indexing and data grouping techniques. So we could measure the effect of our novelty index schema (per-node based) along with the data grouping, we compared ShortMap and an indexed column-oriented store. Then, we evaluate the overall performance of ShortMap, by comparing executions of our prototype of ShortMap with the unmodified version of Hadoop. Our experimental results show that our solution can provide speedups up to 80 times the default Hadoop implementation and that also outperforms those competing solutions. Finally, we have shown that ShortMap is also able to outperform Spark.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions

5.1 Conclusions

We have described the design, implementation, and experimental evaluation of ShortMap, a system that significantly improves the performance of MapReduce jobs that are concerned with just a subset of the entire dataset. Our experimental results, obtained with a sample of a real dataset, show that ShortMap can provide speedups up to 80 times the default Hadoop implementation. Naturally, better results are obtained for queries that target uncommon values, but our results show that ShortMap does not incur performance degradation even when querying for the common attributes; actually, it still provides some (although arguably small) benefits in the less favourable cases. We have also extensively compared ShortMap against other state of the art MapReduce implementations that have materialised techniques similar to ours, although in different forms. Our results show that ShortMap also outperforms those competing solutions. ShortMap has been implemented as open source and has been made available for others to experiment and to improve upon.

5.2 Future Work

As future work we would like to leverage on the insights on frequencies of attribute values available in the indexes to build a better load balancing for ShortMap. We believe this frequency information could be leveraged to increase the size of the splits used by Hadoop for queries with high selectivity, thus reducing the management costs involved in creating Map tasks.

References

Abadi, D. J., S. R. Madden, & N. Hachem (2008). Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, New York, NY, USA, pp. 967–980. ACM.

Ailamaki, A., D. J. DeWitt, M. D. Hill, & M. Skounakis (2001). Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, San Francisco, CA, USA, pp. 169–180. Morgan Kaufmann Publishers Inc.

Aiyer, A. S., M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, & M. Vaidya (2012). Storage infrastructure behind facebook messages: Using hbase at scale. *IEEE Data Eng. Bull. 35*(2), 4–13.

Bertrand, K. Z., M. Bialik, K. Virdee, A. Gros, & Y. Bar-Yam (2013). Sentiment in new york city: A high resolution spatial and temporal view. *arXiv preprint arXiv:1308.5010*.

Dean, J. & S. Ghemawat (2004). Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Dittrich, J., J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, & J. Schad (2010, September). Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow. 3*(1-2), 515–529.

Dittrich, J., J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, & J. Schad (2012). Only aggressive elephants are fast elephants. *CoRR abs/1208.0287*.

Eltabakh, M. Y., F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, & J. Vondrak (2013). Eagle-eyed elephant: Split-oriented indexing in hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, New York, NY, USA, pp. 89–100. ACM.

Ferreira, M., J. Paiva, & L. Rodrigues (2014). Suporte eficiente para pesquisas seletivas em

mapreduce. In *Actas do Sexto Simpósio de Informática*, Inforum'14, Porto, Portugal.

Floratou, A., J. M. Patel, E. J. Shekita, & S. Tata (2011, April). Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow. 4*(7), 419–429.

Frank, M. R., L. Mitchell, P. S. Dodds, & C. M. Danforth (2013). Happiness and the patterns of life: a study of geolocated tweets. *Scientific reports 3*.

Ghemawat, S., H. Gobioff, & S.-T. Leung (2003, October). The google file system. *SIGOPS Oper. Syst. Rev. 37*(5), 29–43.

He, Y., R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, & Z. Xu (2011). Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, Washington, DC, USA, pp. 1199–1208. IEEE Computer Society.

http://hadoop.apache.org.

https://dev.twitter.com/docs/platform objects/tweets.

Jiang, D., B. C. Ooi, L. Shi, & S. Wu (2010, September). The performance of mapreduce: An in-depth study. *Proc. VLDB Endow. 3*(1-2), 472–483.

Jindal, A., J.-A. Quiané-Ruiz, & J. Dittrich (2011). Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, pp. 21:1–21:14. ACM.

Kavulya, S., J. Tan, R. Gandhi, & P. Narasimhan (2010). An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, Washington, DC, USA, pp. 94–103. IEEE Computer Society.

Kloumann, I. M., C. M. Danforth, K. D. Harris, C. A. Bliss, & P. S. Dodds (2012). Positivity of the english language. *PloS one 7*(1), e29484.

Lee, K.-H., Y.-J. Lee, H. Choi, Y. D. Chung, & B. Moon (2012, January). Parallel data processing with mapreduce: a survey. *SIGMOD Rec. 40*(4), 11–20.

Lin, J., D. Ryaboy, & K. Weil (2011). 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, New York, NY, USA, pp. 59–66. ACM.

Lin, M., L. Zhang, A. Wierman, & J. Tan (2013, October). Joint optimization of overlapping phases in mapreduce. *Perform. Eval. 70*(10), 720–735.

Mitchell, L., M. R. Frank, K. D. Harris, P. S. Dodds, & C. M. Danforth (2013). The geography of happiness: Connecting twitter sentiment and expression, demographics, and objective characteristics of place. *PloS one 8*(5), e64417.

Page, L., S. Brin, R. Motwani, & T. Winograd (1999, November). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

Palanisamy, B., A. Singh, L. Liu, & B. Jain (2011). Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, pp. 58:1–58:11. ACM.

Pavlo, A., E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, & M. Stonebraker (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, New York, NY, USA, pp. 165–178. ACM.

Richter, S., J.-A. Quiané-Ruiz, S. Schuh, & J. Dittrich (2012). Towards zero-overhead adaptive indexing in hadoop. *CoRR abs/1212.3480*.

Shvachko, K., H. Kuang, S. Radia, & R. Chansler (2010). The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, Washington, DC, USA, pp. 1–10. IEEE Computer Society.

van der Lande, J. (2013, April). Big data analytics: Telecoms operators can make new revenue by selling data, http://www.analysysmason.com/.

Zaharia, M., M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, & I. Stoica (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, pp. 2–2. USENIX Association.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, & I. Stoica (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics*

in Cloud Computing, HotCloud'10, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Zaharia, M., A. Konwinski, A. D. Joseph, R. Katz, & I. Stoica (2008). Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, Berkeley, CA, USA, pp. 29–42. USENIX Association.

Zipf, G. K. (1935). *The Psychobiology of Language*. Boston, MA: Houghton Mifflin.