Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Matti Niemenmaa

Analysing sequencing data in Hadoop:

The road to interactivity via SQL

Master's Thesis
Espoo, 16th November 2013

Supervisor: Assoc. Prof. Keijo Heljanko
Advisor: Assoc. Prof. Keijo Heljanko

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Matti Niemenmaa

Title:
Analysing sequencing data in Hadoop: The road to interactivity via SQL

Date: 16th November 2013 **Pages:** xv + 143

Major: Theoretical Computer Science **Code:** T-79

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: Assoc. Prof. Keijo Heljanko

Analysis of high volumes of data has always been performed with distributed computing on computer clusters. But due to rapidly increasing data amounts in, for example, DNA sequencing, new approaches to data analysis are needed. Warehouse-scale computing environments with up to tens of thousands of networked nodes may be necessary to solve future Big Data problems related to sequencing data analysis. And to utilize such systems effectively, specialized software is needed.

Hadoop is a collection of software built specifically for Big Data processing, with a core consisting of the Hadoop MapReduce scalable distributed computing platform and the Hadoop Distributed File System, HDFS. This work explains the principles underlying Hadoop MapReduce and HDFS as well as certain prominent higher-level interfaces to them: Pig, Hive, and HBase. An overview of the current state of Hadoop usage in bioinformatics is then provided alongside brief introductions to the Hadoop-BAM and SeqPig projects of the author and his colleagues.

Data analysis tasks are often performed interactively, exploring the data sets at hand in order to familiarize oneself with them in preparation for well targeted long-running computations. Hadoop MapReduce is optimized for throughput instead of latency, making it a poor fit for interactive use. This Thesis presents two high-level alternatives designed especially with interactive data analysis in mind: Shark and Impala, both of which are Hive-compatible SQL-based systems.

Aside from the computational framework used, the format in which the data sets are stored can greatly affect analytical performance. Thus new file formats are being developed to better cope with the needs of modern and future Big Data sets. This work analyses the current state of the art storage formats used in the worlds of bioinformatics and Hadoop.

Finally, this Thesis presents the results of experiments performed by the author with the goal of understanding how well the landscape of available frameworks and storage formats can tackle interactive sequencing data analysis tasks.

Keywords: Hive, Shark, Impala, Hadoop, MapReduce, HDFS, SQL, sequencing data, Big Data, interactive analysis

Language: English

Acknowledgements

To my supervisor and my colleagues at work, for the valuable feedback on the content of this Thesis and for teaching me some things I needed to know.

To my friends at Aalto, for the stimulating lunchtime discussions.

To my family and girlfriend, for your support and patience.

Espoo, 16th November 2013

Matti Niemenmaa

Contents

Contents vii

List of abbreviations ix

List of Tables xiii

List of Figures xiv

List of Listings xv

1 Introduction 1

2 MapReduce 7
2.1 Execution model 9
2.2 Distributed file system 13

3 Apache Hadoop 15
3.1 Apache Pig 17
3.2 Apache Hive 19
3.3 Apache HBase 22

4 Hadoop in bioinformatics 27
4.1 Hadoop-BAM 28
4.2 SeqPig 30

5 Interactivity 33
5.1 Apache Spark 34
5.2 Shark 37

5.3 Cloudera Impala 39

6 Storage formats **41**
6.1 Row-oriented binary storage formats 44
Compression schemes 45
BAM and BCF 46
Considerations for bioinformatical file format design . . . 49
6.2 RCFile 50
6.3 ORC 51
6.4 Trevni 52
6.5 Parquet 53

7 Experimental procedure **55**
7.1 Accessing sequencing data 55
7.2 Intended procedure 60
7.3 Issues encountered 61
7.4 Final procedure 63
7.5 Setup 69

8 Experimental results **73**
8.1 Data set size 73
8.2 Query performance 75
Overviews by framework 75
Overviews by storage format 78
A closer look at speedups 84
Detailed comparisons 87

9 Conclusions **99**

A Experimental configuration **103**
A.1 Hadoop 103
A.2 Hive 105
A.3 Shark 105
A.4 Impala 106

B HiveQL statements used **109**
B.1 Table creation and settings 109
B.2 Queries on the full data set 111
B.3 Exploratory queries on the reduced data set 113

Bibliography **117**

List of abbreviations

Throughout this work, a *byte* represents an eight-bit quantity.

API	application programming interface
BAM	Binary Alignment/Map [Li+09; SAM13]
bp	base pair
BCF	Binary Call Format [Dan+11]
BED	Browser Extensible Data [Qui+10]
BGZF	Blocked GNU Zip Format *(according to e.g. Cánovas and Moffat [Cán+13] and Cock [Coc11])*
BGI	华大基因, *a Chinese genomics research institute; formerly Beijing Genomics Institute*
CDH	Cloudera's Distribution Including Apache Hadoop [CDH]
CIGAR	Compact Idiosyncratic Gapped Alignment Report [Ens13]
CPU	central processing unit
CRC	cyclic redundancy check [Pet+61]
DAG	directed acyclic graph
DDR3	double data rate [DDR08], type three
DEFLATE	*a compressed data format, or the canonical compression algorithm outputting data in that format [Deu96a]*
DistCp	distributed copy, *a file copying tool using Hadoop Map-reduce* [DCp]
DNA	deoxyribonucleic acid
DOI	Digital Object Identifier [DOI]
ETL	Extract, Transform, and Load
GATK	the Genome Analysis Toolkit [McK+10]
GB	gigabytes (10^9 bytes)
Gbps	gigabits per second (10^9 bits per second)
GFS	the Google File System [Ghe+03]

GHz	gigahertz (10^9 Hertz)
GiB	gibibytes (2^{30} bytes)
GNU	GNU's Not Unix! [GNU]
HDFS	the Hadoop Distributed File System
HiveQL	the Hive query language *(in this work, also used to refer to the dialects understood by Shark and Impala)*
HTS	high-throughput sequencing
I/O	input/output
JBOD	just a bunch of disks
JVM	Java Virtual Machine
kB	kilobytes (10^3 bytes)
KiB	kibibytes (2^{10} bytes)
LLVM	*a collection of compiler-related software projects [LLV]; formerly Low-Level Virtual Machine [Lat+04]*
LZMA	Lempel-Ziv-Markov chain algorithm [Pav13]
LZO	Lempel-Ziv-Oberhumer [Obe]
MB	megabytes (10^6 bytes)
MHz	megahertz (10^6 Hertz)
MiB	mebibytes (2^{20} bytes)
Mibp	mebi-base pairs (2^{20} base pairs)
MPI	Message Passing Interface [MPI93]
MTBF	mean time between failures
N/A	not applicable
NFS	Network File System [Sto+10]
NGS	next-generation sequencing
ORC	Optimized Row Columnar [ORC13; ORM]
PB	petabytes (10^{15} bytes)
PiB	pebibytes (2^{50} bytes)
PNG	Portable Network Graphics [Duc03]
QC	quality control
QDR	quad data rate
RAM	random access memory
RCFile	Record Columnar File [He+11]
RDD	Resilient Distributed Dataset [Zah+12]
RPM	revolutions per minute
SAM	Sequence Alignment/Map [Li+09; SAM13]
SDRAM	synchronous dynamic random access memory [SDR94]
SerDe	serializer/deserializer
SQL	Structured Query Language [ISO92]
SSTable	Sorted String Table [McK+09]
stddev	standard deviation

TB	terabytes (10^{12} bytes)
TiB	tebibytes (2^{40} bytes)
URL	Uniform Resource Locator [Ber+05]
UTF	Unicode Transformation Format [UTF]
VCF	Variant Call Format [Dan+11]
XML	Extensible Markup Language [Bra+08]
YARN	Yet Another Resource Negotiator [Wat12]

List of Tables

6.1 BCF record format. 48

7.1 BAM record format. 57
7.2 Hive schema used for BAM data. 59
7.3 Data set size initially and after each modification. 68
7.4 The experiment plan. 70
7.5 Unfinished 31-worker experiments. 72

8.1 The data set size in different formats. 74
8.2 `gzip` vs. Snappy runtimes with Hive and RCFile. 90
8.3 BAM vs. `gzip`-compressed RCFile runtimes with Hive. 91
8.4 Hive runtimes with `gzip`-compressed RCFile vs. DEFLATE-compressed ORC. 92
8.5 RCFile vs. ORC runtimes with Hive and Snappy compression. 93
8.6 Impala vs. Shark runtimes on a `gzip`-compressed RCFile `bam` table. 94
8.7 Runtimes of Impala with Snappy-compressed Parquet vs. Shark with `gzip`-compressed RCFile. 96

A.1 Hadoop settings given in `core-site.xml`. 104
A.2 HDFS settings given in `hdfs-site.xml`. 104
A.3 Hadoop MapReduce settings given in `mapred-site.xml`. . . 105
A.4 Relevant environment variables for Hadoop. 105
A.5 Hive configuration variables. 106
A.6 Shark environment variables. 106
A.7 Parameters given in `SPARK_JAVA_OPTS`. 106
A.8 Impala environment variables, all concerning only logging. . . 107

List of Figures

1.1 Historical trends in storage prices vs. DNA sequencing costs. . 3

2.1 Distributed MapReduce execution. 10

3.1 HBase state and operations. 25

4.1 Speedup observed in BAM sorting with Hadoop-BAM. 30
4.2 Speedup of SeqPig vs. FastQC. 31

8.1 Query times on a linear scale, by framework. 76
8.2 Query times on a log scale, by framework. 77
8.3 Impala's `bam` query times on a linear scale, by table format. . . 79
8.4 Hive's `bam` query times on a linear scale, by table format. . . . 80
8.5 Hive's `bam` query times on a log scale, by table format. 81
8.6 Hive's `results` query times on a linear scale, by table format. 82
8.7 Hive's `results` query times on a log scale, by table format. . 83
8.8 Hive's post-BED join query times on a linear scale, by table format. 85
8.9 Shark's `bam` query times on a linear scale, by table format. . . 86
8.10 Shark's `bam` query times on a log scale, by table format. 87
8.11 Shark's post-BED join query times on a linear scale. 88
8.12 Impala's post-BED join query times on a linear scale. 89

List of Listings

7.1 HiveQL initializing RCFile with `gzip` for Hive. 64
7.2 HiveQL used on the `bam` table in Hive. 65
7.3 HiveQL used on the `results` table in Hive. 67
B.1 HiveQL code describing the table schema. 109
B.2 HiveQL used to create `bam` in BAM. 110
B.3 HiveQL used to create `bam` in RCFile. 110
B.4 HiveQL used to create `bam` in ORC. 110
B.5 HiveQL used to create `bam` in Parquet. 110
B.6 HiveQL used to create the BED table. 110
B.7 RCFile compression settings used with both compressors. . 110
B.8 The RCFile `gzip` compression setting. 111
B.9 The RCFile Snappy compression setting. 111
B.10 HiveQL initializing post-BED join benchmarking in Shark. 111
B.11 HiveQL initializing single-node post-BED join benchmarking in Impala. 111
B.12 The parallelism setting for Hive and Shark. 111
B.13 Initial counting statements on the full data set. 111
B.14 Statements computing the two histograms. 112
B.15 Code specifying the columns in the `bam` table. 112
B.16 The join with the BED table. 112
B.17 HiveQL copying the separately computed BED join data. . 113
B.18 HiveQL counting the size of the result of the BED join. . . 113
B.19 HiveQL computing the quality join and its size. 113
B.20 Simple filters and interspersed counts on `results`. 114
B.21 HiveQL calculating the mean and standard deviation. . . . 114

Chapter 1

Introduction

> [A] wealth of information creates a poverty of attention, and a need to allocate that attention efficiently among the overabundance of information sources that might consume it.
>
> 'Designing Organizations for an Information-Rich World'
> Herbert A. Simon [Sim71]

Data volumes nowadays are increasing to the point that many individual data sets are too large to be analysed, or even stored, on a single computer. Such data sets are known as *Big Data*, and can arise in several contexts. Examples include Internet searches, financial analytics, and various fields of science. Notably many Big Data problems can be found in the field of bioinformatics. A number of them are due to recent advances in *sequencing*: the task of determining the base composition of e.g. DNA, possibly going as far as finding the entire genome of an organism.

In the case of DNA, the number of *base pairs* or *bp*, the building blocks of genomic information, that can be sequenced per unit cost has been growing at an exponential rate for over two decades, doubling approximately every 19 months [Ste10]. This alone would have caused Big Data issues sooner or later. However, the growth rate suddenly increased around the year 2005, due to the emergence of techniques known as *high-throughput sequencing* or *HTS* (a.k.a. *next-generation sequencing* or *NGS*). HTS has resulted in the process speeding up to the point that the cost has now been halving approximately every five months [Ste10]. As an example of current speeds, Pireddu, Leo, and Zanetti [Pir+11a] claim that their 'medium-sized' DNA sequencing laboratory can create 4–5 TB of data every week. At the high

end, BGI, 'one of the largest producers of genomic data in the world', generates 6 TB of data daily [Mar13]. For comparison, the largest hard disk drives available as of November 2013 are 6 TB in size [HGS13].

Exponential growth due to technological advances is not unusual in the computing world. Consider the following three 'laws':

- Moore's law: the number of components in integrated circuits with minimum cost per component doubles every year [Moo65]. Later amended to a doubling every two years without the minimum cost aspect [Moo75], and commonly quoted as 18 months [Int05]. Together with Dennard scaling [Den+07], Moore's law has meant that processing power has doubled at essentially the same rate.[1]

- Butters' law (of photonics): the cost of transmitting one bit over an optical network halves every nine months [Rey98].

- Kryder's law, which was never given as a prediction, merely an observation: areal storage density of hard disk drives had been increasing at a greater rate than the rate of processor improvement according to Moore's law [Wal05].

Note, however, that none of the above growth rates, corresponding respectively to increases in processing power, network speed, and storage capacity, are even close to as fast as the pace at which sequencing is currently improving. See Figure 1.1 for a clarifying plot comparing trends in storage and sequencing costs from 1990 to 2009. (For comparing the actual values instead of only the overall trends, one must know the size of a base pair, which depends on the storage format: for example, a single base is stored in 4 bits in BAM files and 8 bits in SAM files [SAM13], excluding compression.) Note that the source of the plot describes Kryder's law as a doubling every 14 months, significantly more optimistic than more recent studies showing that the period is about 25 months [Kry+09]. Nevertheless, storing sequencing data on a hard disk is, or will soon be, actually more expensive than generating the data [Ste10], making its storage an increasingly difficult task. Discarding all but the most informative parts may be the only long-term option.

Small enough data sets may fit completely in main memory, enabling computation that may be faster (per unit of size) than on larger sets by

[1] But the end of Dennard scaling made improving single-core CPUs much more difficult than it was previously, leading manufacturers to turn to multi-core designs instead [Esm+11; Sut09]. Furthermore, there are signs that multi-core scaling will also not last long [Bos13; Esm+11; Har+11].

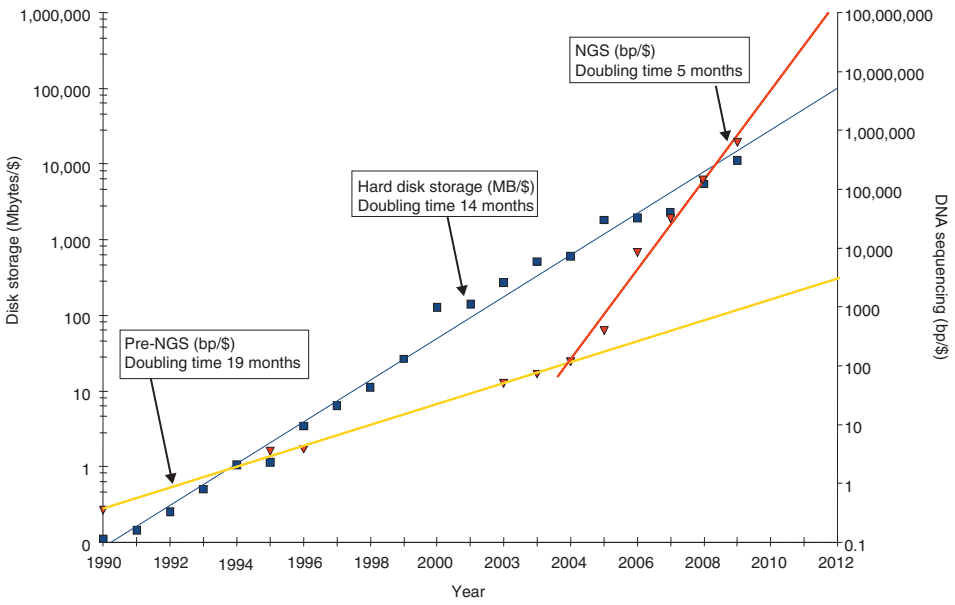

Figure 1.1: Historical trends in storage prices vs. DNA sequencing costs. Reproduced from Stein [Ste10].

several orders of magnitude. Technological progress may result in this applying to current Big Data sets in as few as ten years: if current trends are followed, by that time the price of RAM (random access memory) will equal the current price of disk storage [Pla+11]. Thereafter many data sets that are currently considered Big Data may be able to be held fully in memory, with disk serving only as backup, in systems such as RAMCloud [Ous+11]. The speed at which e.g. sequencing data grows prevents such systems from being complete solutions to the problem of efficiently processing Big Data, but they can be very effective for data sets that are not overly large.

Storage feasibility is only part of the picture: like any kind of raw data, sequencing data also needs to be analysed in order for it to be of any use. Clearly, if there is too much data for even its storage to be possible, its analysis is equally infeasible. This magnitude of data classifies sequence data analysis as a Big Data problem.

For a problem to qualify as a Big Data problem, attempting to solve it with a single computer should result in one or both of the following:

- The computer has too slow a processor or too little memory to be able to perform the needed computations in a reasonable amount of time. Waiting for better hardware will not help, because the data growth outpaces Moore's law.

- The computer does not have enough disk space to store the data sets on which computations are to be performed. Waiting for larger disk drives will not help, because the data growth outpaces Kryder's law.

Therefore, in order to solve Big Data problems, lone computers are insufficient. *Distributed* computing is required, i.e. having multiple networked computers, or *nodes*, working together in computer *clusters*. Ideally, the clusters used have been specialized for the task at hand, thus making them effectively *warehouse-scale computers* [Bar+13].

Traditionally, distributed software has been created by developing communication protocols specific to the application, using primitives provided by e.g. MPI (the Message Passing Interface) [MPI93], the PVM framework [Sun90] or, in the data communications domain, the Erlang programming language [Arm97]. At this level, implementing the necessary functionality correctly is difficult, especially if the software is to be run not only in small clusters but on warehouse-scale computers, with hundreds to tens of thousands of network nodes. Realizing high performance in such an environment is especially complicated. In addition, fault tolerance becomes a necessity, because the probability of hardware failure is too high to ignore [Dea09].

To ensure that warehouse-scale distributed software can work at high performance and not worry about hardware failure, a framework specifically designed for that use case is necessary. One such framework was developed by Google [Goo]: *MapReduce* [Dea+04] coupled with *GFS* (the Google File System) [Ghe+03]. Together, they provide fault tolerance both for computations and data: most hardware failures neither interrupt running processes nor cause data loss.

The implementations of the MapReduce system and GFS were not made publicly accessible, leading to the creation of *Apache Hadoop* [Had], an open source implementation of the same ideas. Hadoop has since expanded to become a collection of software related to scalable distributed computing.

Unfortunately, there exist problems for which MapReduce's computational model is far from ideal. In particular, MapReduce is specifically optimized for throughput over latency, which makes it a poor fit for *interactive* use. Interactive analysis tasks arise e.g. when users are not well enough acquainted with the data sets concerned to effectively perform long-running computations on them, having to instead *explore* them with repeated queries, either narrowing down areas of interest or requesting more information according to newly realized needs [Hee+12]. MapReduce's typical ten-second job startup time [Pav+09; Xin+12] guarantees that most users will shift their focus before a computation completes [Car+91], slowing

down this exploratory process. Interactive tasks are increasingly prevalent in sequencing data analysis [Che+12], making frameworks designed with latency in mind desirable. Low latency is a more difficult goal to reach than high throughput [Dea+13; Pat04], but nevertheless such systems do exist. Two notable ones, whose performance is evaluated in this work, are Shark [Xin+12], which is based on Apache Spark [Zah+12], and Cloudera Impala [Imp], which is based on the design of Google's Dremel [Mel+10]. Both frameworks allow access to structured data stored in HDFS (the Hadoop Distributed File System) using a language based on SQL (Structured Query Language) [Cha+74; Gro+09; ISO92], contributing to the trend [Mur+13] of handling interactive Big Data computations with 'SQL-on-Hadoop'.

This Thesis proceeds as follows. In Chapter 2 the background of MapReduce as well as its specifics, especially pertaining to Hadoop, are gone over in detail. Next, Chapter 3 covers the Hadoop project and some notable high-level frameworks based on it: Apache Pig [Ols+08], the SQL-based Apache Hive [Thu+10a], and Apache HBase [HBa]. Chapter 4 surveys the current state of Hadoop in bioinformatics and presents two sets of tools developed by the author and his colleagues that enable using Hadoop to manipulate and analyse sequencing data. Chapter 5 delves into the interactivity-oriented Shark and Impala frameworks. Chapter 6 discusses the importance of storage formats and studies the current state of the art formats in the worlds of sequencing and SQL-on-Hadoop. With the necessities covered, Chapter 7 presents a set of benchmarks used to compare the effectiveness of the exhibited SQL-based frameworks—Hive, Shark, and Impala—in interactive sequencing data analysis. The results obtained by running the benchmarks are examined in Chapter 8. Finally, Chapter 9 states some final thoughts based on the results of the experiments and the current state of scalable interactive sequencing data processing.

Chapter 2

MapReduce

> A computer's attention span is as long as its power cord.
>
> unknown

Applying warehouse-scale computing to Big Data problems is not as simple as setting up the hardware. Programming for a warehouse-scale computer is a far more complex task than programming for a small cluster, which in itself is more challenging than programming for e.g. a typical desktop system. This is especially the case when performance is a concern, since effectively utilizing all available resources involves co-ordinating several hardware and software layers. Examples of things to keep in mind are the complex memory hierarchy, heterogeneous hardware, failure-prone components, and network topology [Bar+13]: all in addition to the complexity of implementing the core of the application itself. As such it is no surprise that programming frameworks that ease the burden on the developer of warehouse-scale applications have been created. MapReduce [Dea+04] is one such framework, including automatic handling for data distribution and fault tolerance.

In order for a warehouse-scale computing framework to be practical, it must be able to tolerate hardware failure. Even with unrealistically reliable servers with a mean time between failures (MTBF) of 30 years, if there are 10 000 servers in a cluster, it will experience on average one failure every day [Bar+13]. This makes fault tolerance in software not only useful, but a practical necessity. In addition, it allows for a better price/performance tradeoff by using relatively unreliable, cheaper hardware [Bar+03]. MapReduce has been designed with this in mind: it provides efficiently fault

tolerant computations and is intended to be used together with file systems that provide fault tolerant data storage.

At its simplest level, MapReduce is a programming model for transforming data: the programmer need only specify two functions—the *Map* and *Reduce* functions—and the input data, and based on this information the corresponding output can be computed in a functional manner. Because of this, the model also allows for a simple strategy for fault tolerance: as executing a function with a given input will always result in the same output, e.g. all computations on a failed computer can trivially be re-executed on another computer, as long as the input data is still available. The MapReduce model allows for easy parallelization and is relatively simple to program for, making it an attractive choice for distributed computing.

However, the term 'MapReduce' in a distributed computing context is generally understood as meaning more than just the abstract programming model: it includes the associated implementation that handles scheduling the computation efficiently and dealing with machine failures during execution.

The original MapReduce implementation [Dea+04] was developed internally at Google and has not been released to the public. The current *de facto* open source implementation of MapReduce is Apache Hadoop [Had], which will be discussed in Chapter 3. Hadoop's existence makes MapReduce an attractive choice as a distributed computation model because Hadoop is well established, having seen use in a variety of fields with good results. (See Chapter 3 for detailed information.)

MapReduce is not perfect, though: its programming model can be considered too rigid for various tasks. For example, PACT [Ale+11] has been explicitly designed as an extension of MapReduce with the ability to express more complex operations. Another example, Spark [Zah+12], instead emphasizes data re-use: MapReduce does not intrinsically allow re-using intermediate results. If such re-use is desired, it must be done by manually saving and loading the corresponding data, which can incur needless I/O and serialization overheads. And of course, as previously mentioned (and elaborated on in Chapter 5), MapReduce is a very poor fit for interactive work. In spite of these limitations, however, the MapReduce model continues to see use across a wide variety of applications.

In the following Sections the MapReduce execution model is discussed in detail before delving into the file systems that MapReduce is typically paired with. The information on MapReduce is based completely on Dean and Ghemawat [Dea+04] and White [Whi09].

2.1 Execution model

Conceptually, the execution model of MapReduce consists only of applying the *Reduce* function to the grouped results of the *Map* function. However, practical distributed MapReduce frameworks complicate the process: they specify more steps and implement them in certain ways to ensure that good performance and fault tolerance are achieved. Below, the simpler, conceptual model is first briefly explained, and then the principles that underlie the warehouse-scale implementations are considered.

The type signatures of the two user-specified functions form a concise description of the conceptual MapReduce execution model. See the following, where k is short for 'key' and v for 'value', the subscripts serve to differentiate the types, and the superscripts $m \geq 0$, $n > 0$, and $p \geq 0$ denote differing list lengths:

$$\begin{aligned} \mathit{Map} &: (k_1, v_1) \rightarrow (k_2, v_2)^m \\ \mathit{Reduce} &: (k_2, v_2^n) \rightarrow v_3^p \end{aligned}$$

As can be deduced from the type signatures, a MapReduce computation takes a sequence of key-value pairs as input, on which it performs the following tasks:

1. The *Map* function is applied to each key-value pair in the input, outputting any number of new key-value pairs for each one.

2. Each key in the output from the previous step is paired with all the values that were associated with that key.

3. The *Reduce* function is applied to each pair in the result of the pairing in the previous step. The resulting list of data forms the final output.

To clarify the process, consider the following simple example, where the task consists of taking as input a set of documents and outputting, for each word encountered, the set of documents it was found in. Here the input type could be e.g. $(k_1, v_1) = (\texttt{document-name}, \texttt{contents})$ for each document. The *Map* function would go through the contents, outputting pairs of type $(k_2, v_2) = (\texttt{word}, \texttt{document-name})$. Thus the *Reduce* function receives as input pairs of the form $(\texttt{word}, \texttt{document-name}^n)$, which is precisely what was desired in the problem statement. The final output v_3^p would depend on the exact format in which the output is desired, but could be e.g. a string (just one string, i.e. $p = 1$) of the form `"word"`, `"document-1-name"`, `"document-2-name"`, ... for each word.

While the above description is sufficient for implementing a basic MapReduce framework, fully distributed systems for warehouse-scale computers, such as Google's MapReduce implementation and Hadoop MapReduce, are more complex and perform the steps in a very specific way. See Figure 2.1 for a graphical overview of how MapReduce computations are executed on such systems.

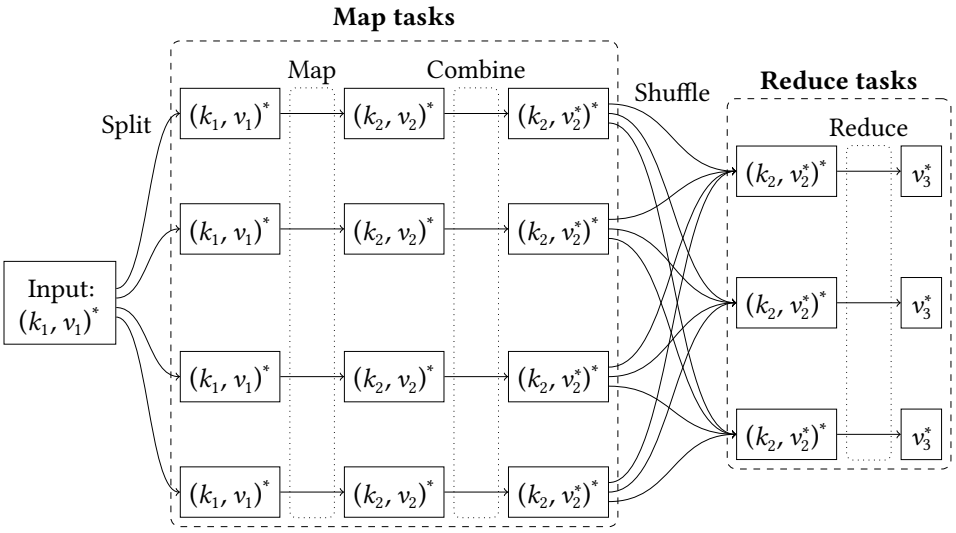

Figure 2.1: Distributed MapReduce execution with four map tasks and three reduce tasks. k_i and v_j denote key type i and value type j respectively. The asterisk superscripts denote unknown list lengths.

Distributed MapReduce is structured as a master-slave system. The master node (known in Hadoop as the *jobtracker* and predefined as a specific node for the whole cluster) allocates workers for different parts of the computation and co-ordinates communication between them. The slaves (known in Hadoop as *tasktrackers*) are the nodes that actually read the input data, run the *Map* and *Reduce* functions, and write the output data. Each slave node provides a number of map and reduce *slots* for running the two different functions. For a computation or *job*, typically the user selects the number of reduce tasks to be performed while the MapReduce system automatically determines the number of map tasks. The full execution process is as follows:

1. **Split** This step is performed solely on the master. The input files are conceptually split into chunks: a set of *splits*, i.e. tuples that identify

sequential parts in the input files, is created. These splits are typically tens of megabytes in size, often corresponding to the block size of the file system in use (see Section 2.2). Based on this the master creates a map task for each split and assigns as many of them as it can to separate map slots, which are started up and begin running. (The remaining tasks are started as the job progresses: when a task completes it frees its slot for use by another task.)

2. **Map** Each map task involves reading the corresponding input split, forming key-value pairs of the data therein, and handing them to the *Map* function for processing. These intermediate key-value pairs are written to local disk, sorted by key, and *partitioned*: differentiated based on which reduce task they belong to. The default partitioning simply assigns each key k to the reducer $h(k) \bmod R$ where h is a hash function [Knu73] and R is the number of reduce tasks.

3. **Combine** This is an optional step that essentially runs the *Reduce* function on the partitioned output of the *Map* function directly as part of the map task. While a custom *Combine* function can be given, typically *Reduce* is used as-is. This use requires that it be commutative and associative. Note that since combining can reduce the map task's output size, it is performed before writing the partitions to local disk, as long as the task has enough available memory for in-memory sorting and partitioning. This way, fewer I/O operations are performed.

4. **Shuffle** The map tasks communicate the locations of their partitioned outputs to the master node. It then notifies the corresponding reduce tasks (starting them up in reduce slots as required) that new data is available. The reduce tasks read the data from the local disks of the nodes where the data was written—note that this may be the same node on which the reduce task itself is running, in which case no network communication is required. When a reduce task has received all of its input data, it sorts it so that it is grouped by key.

5. **Reduce** Each reduce task iterates over its sorted sequence of key-value pairs, passing each unique key and corresponding sequence of values to the *Reduce* function. The output from it is written directly to the output file of the reduce task, which is one of the final output files generated by the MapReduce computation.

The end result is a set of output files, one from each reduce task. They are not automatically combined to a single file because that is not always

necessary: they could be used as-is as inputs for another MapReduce job, for example. It is also possible to run a *map-only job* in which only the *Map* function is used, with the map tasks' output forming the output for the entire computation.

Fault tolerance in this kind of a fully distributed MapReduce system is fairly simple to implement. The master node periodically pings the slaves, assuming them to have failed if it does not receive a response in time. In-progress tasks on failed nodes are rescheduled and eventually restarted. Completed map tasks are also rescheduled, but completed reduce tasks are not. This is because the input and output files are assumed to be on a shared storage system, separate from the local disks that are used for storing the intermediate output from map tasks. Thus, if a node with a completed map task whose output has not yet been sent to a reduce task fails, the map task needs to be restarted, but if a node with a completed reduce task fails, nothing needs to be done. This way worker failure is fully accounted for, which is important for long-running jobs at warehouse scale. In contrast, master failure is deemed unlikely since it requires a specific node to fail, and is not handled at all, making the master a single point of failure.

Sometimes worker nodes may have unexpectedly poor performance due to e.g. faulty hardware. This results in *stragglers*: members of the last few map or reduce tasks which take a particularly long time to complete, holding up the whole computation. A key optimization in MapReduce systems, that of *speculative* or *backup* execution, was designed to mitigate this problem. After all tasks have been started, if some tasks have been running for a relatively long time and seem to be progressing (performing I/O of key-value pairs) relatively slowly, the master attempts to reschedule those same tasks on different nodes. When a task is successfully completed, any other executing duplicates of that task are stopped. Speculative execution does not significantly affect the resources used by a job but can speed it up greatly.

Since tasks can run multiple times as well as be restarted at any point, the *Map* and *Reduce* (and *Combine*, if used and distinct from *Reduce*) functions should be free of side effects: *pure* functions of their input values. Only then is it guaranteed that all the output of a fully distributed MapReduce system is equivalent to a single sequential execution of the program. In the face of nondeterministic user-supplied functions, the output of each reduce task may correspond to a different sequential execution. Whether this inconsistency is a problem in practice depends on the application.

2.2 Distributed file system

MapReduce is traditionally paired with a specific distributed file system, designed for large files and streaming access patterns. For Google's MapReduce that file system is GFS (the Google File System) and for Hadoop it is HDFS (the Hadoop Distributed File System). Both share similar design principles and implementation strategies, which will be covered in the remainder of this Section. Information on GFS in this Section is based on Ghemawat, Gobioff, and Leung [Ghe+03] and information on HDFS is based on White [Whi09], except where otherwise indicated.

GFS and HDFS are both, like MapReduce, master-slave systems. The master node (known in Hadoop as the *namenode*) keeps track of the state of the slaves as well as file metadata, and the slave nodes (known in Hadoop as the *datanodes*) are responsible for all data storage and communication. Replication is used to provide fault tolerance: each block is stored on multiple slaves—three by default. For simplicity reasons [McK+09] the master node is a single point of failure, though HDFS's *secondary namenode* can limit data loss in case of catastrophic master node failure.

When using MapReduce, the slave nodes should be used to run MapReduce workers as well, allowing MapReduce to take advantage of data locality for map tasks. This is done by scheduling map tasks on nodes where the data for that task's split is stored, or, failing that, on nodes that are nearby in terms of the network topology. Replication is advantageous here as well as for fault tolerance, since it improves the odds of being able to schedule a task on a node that has the corresponding split's data available locally. Note that it is possible to run a MapReduce job on GFS and HDFS without any of the input data being sent across the network.

A major design principle of both GFS and HDFS is to support large files efficiently. 'Large' in this context means at least 100 megabytes, but typically several gigabytes, and up to terabytes. In contrast, small files are assumed to be rare, and so are not optimized for at all. This is very much the opposite of what file systems are traditionally optimized for [Gia99], which is one of the main reasons that GFS and HDFS are typically paired with MapReduce; they are both intended for Big Data sets consisting of large files.

Another important design principle of GFS and HDFS is the emphasis on *write-once, read-many* operation and streaming reads: written files are assumed to be modified rarely if at all, and workloads are expected to include reading entire files or at least significant portions of them. Random reads and writes are not optimized for—in fact, HDFS does not support random writes at all. This lack of arbitrary modifications makes implementing

replication much simpler, and the philosophy of large reads makes bandwidth far more important than latency. Once again, this ties in with the way MapReduce works, but it is also a more generally helpful restriction for scalable storage architectures: for example, the lowest layer of the Windows Azure Storage [Cal+11; WAz] system has the same limitation.

A notable result of these design decisions is that the block size of both GFS and HDFS is unusually large: 64 MiB. (HDFS does allow changing this, but reducing it to usual file system block sizes would be self-defeating.) This reduces overhead related to metadata management, mainly by drastically reducing the amount of metadata: compared to a more traditional 4 KiB block size and assuming a large enough file, 16 384 times less blocks have to be kept track of. Thus metadata can be kept fully in the memory of the master, making metadata operations fast and enabling easy rebalancing (replica distribution) and garbage collection. Keeping metadata in memory has a drawback, however: it makes the capabilities of the master limit the number of files that can be stored [McK+09]. Another benefit of large block sizes is that if the time to read a full block is much greater than the physical seek time of the disk drives used, reading a file consisting of multiple arbitrarily distributed blocks operates at a speed close to the drives' sequential read rate.

Chapter 3

Apache Hadoop

First, solve the problem. Then, write the code.

unknown

Apache Hadoop [Had; Whi09] was originally conceived as a nameless part of the Nutch [Nut] Web search engine, implementing open source versions of MapReduce [Dea+04] and GFS (the Google File System) [Ghe+03] for its own purposes, in the Java programming language [Gos+13]. Yahoo! [Ya!] soon began contributing to the project, at which point these components were separated, forming the Hadoop project, named after the creator Doug Cutting's child's toy elephant. At around the same time, Hadoop began to be hosted by the Apache Software Foundation [Apa], giving it the full name 'Apache Hadoop'. Since then, Hadoop has grown to become a collection of related projects, two of which are the original MapReduce and file system components: Hadoop MapReduce and HDFS (the Hadoop Distributed File System).

For most of Hadoop's history, the MapReduce component has been the only computational framework supported in Hadoop. Tasks running on other systems, e.g. MPI [MPI93], have not been able to be scheduled on Hadoop clusters. This has meant that the machines in a cluster should be configured to run only one class of tasks, such as Hadoop MapReduce jobs or MPI processes. Otherwise, one node may have several computationally intensive tasks running at once, possibly resulting in resource starvation issues such as running low on memory or disk space, which may in turn cause all tasks on the node to fail. On the other hand, the traditional solution of partitioning the cluster by framework can lead to poor resource utilization, with some machines remaining completely idle while there is work

to do, just because they have been configured for a different framework. Apache Mesos [Hin+11; Mes] is a cluster manager with cross-framework scheduling, solving this problem more effectively. The latest releases of Hadoop (the 2.x series) include their own similar system, called *YARN* (Yet Another Resource Negotiator [Wat12]) [Mur12; YRN; YRN13], also known as NextGen MapReduce or MapReduce 2.0. In addition to cross-framework scheduling, YARN also removes the concept of map and reduce slots from MapReduce slave nodes, instead dynamically allocating map and reduce tasks according to what is most needed at the time. YARN takes over some of the cluster management responsibilities currently handled by Hadoop MapReduce, allowing other computational frameworks to effectively co-exist within Hadoop.

Several companies provide their own distributions of Hadoop, for which they also naturally offer commercial support. The most notable such companies are Cloudera [Clo], Hortonworks [Hor], and MapR [MaR]. They are naturally all major contributors to Hadoop, but have their own extensions as well. Hortonworks's distribution is the only one with support for running on Windows Server. Their contributions are also particularly noteworthy for their *Stinger Initiative* [StI], which involves improving the performance of the Hive project, which is presented in Section 3.2. Cloudera Impala [Imp] is a distributed query engine meant for interactive use, as opposed to MapReduce's emphasis on throughput, and is further discussed in Chapter 5. MapR's distribution provides fault tolerance for the master in both MapReduce and HDFS: the jobtracker is restarted on failure and the namenode is fully distributed. MapR is also unique in that it does not use HDFS; a complete rewrite in the C++ programming language [Str13], whose interface is nevertheless compatible with HDFS, is used instead.

Usage of Hadoop within an organization is unlikely to encompass the entire range of Hadoop-related projects. Some may not even use the MapReduce component, due to the existence of other computational engines and YARN. One thing, however, is common to almost all users of any part of Hadoop: HDFS. The amount of data an organization has stored in HDFS is an indication both of how much the organization uses Hadoop and of what kinds of data volumes Hadoop has been used for. For demonstration purposes, the following is a sample of HDFS usage:

- As of 2013, Facebook [Fac] stores more than 300 PB of data 'in a few large Hadoop/HDFS-based clusters' [Pre; Tra13]. In 2010, they stored 15 PB of data with 60 TB being added daily, and with compression reducing the space usage to 2.5 PB and 10 TB respectively [Thu+10b]. Clearly the rate at which data is added has increased since then, or

they would not have reached the 300 PB mark yet.

- In 2010, Yahoo! had over 82 PB of data among over 25 000 servers split into clusters of about 4000 [Ree10].
- In 2010, Twitter [Twi] had '(soon) PBs of data', with 7 TB of new data coming in every day [Wei10].

Chapter 2 already detailed MapReduce and HDFS. In the following Sections, three prominent open source projects related to Hadoop are instead discussed. Each offers its own higher-level abstraction on top of Hadoop MapReduce and HDFS. Apache Pig offers a high-level language for expressing MapReduce programs, Apache Hive provides a data management and querying system using an SQL-like language implemented with MapReduce, and Apache HBase allows scalable random access into a key-value store in HDFS.

3.1 Apache Pig

Apache Pig [Ols+08; Pig], originally developed by Yahoo!, is a high-level interface to MapReduce, providing a custom query language for bulk data manipulation called *Pig Latin*. Pig Latin is compiled into a sequence of MapReduce computations which are executed on Hadoop. Pig drastically lowers the bar of using Hadoop MapReduce, giving users a richer pool of primitives they can use to describe their computations and not requiring them to implement it in Java, a far more low-level programming language than Pig Latin. This can greatly simplify development and maintenance, improving programmer productivity. Similarly, Pig can be used as a high-level way of implementing what is known as an ETL (Extract, Transform, and Load) pipeline [Sha+12].

Pig treats all data as *relations*. Relations are defined as bags (a.k.a. multisets) of *tuples*. The fields in the tuples can be simple values like integers or strings, but also complex like key-value mappings or even other bags and tuples—arbitrary nesting is allowed. Tuples in a relation are not constrained in any way: they can have different numbers of fields as well as different field types in the same position. It is possible, however, to define a *schema* which specifies a common type for the tuples in a relation. Without a schema, Pig infers a 'safe' type for every field (such as double-width floating point for all numbers), which can cause performance to suffer.

The data model is similar to that used by traditional relational database systems [Cod70] but more flexible. The lack of a defined ordering is

particularly useful for MapReduce processing, as it does not restrict the partitioning strategy (how map outputs are spread among the reducers) in any way. In addition, allowing arbitrary nesting can simplify operations compared to only having flat tables, especially if they are normalized [Dat06], since all data can be kept in one relation instead of having to perform join operations when needed.

Pig Latin has several *commands* for working with relations, and more are being added as development proceeds. The following list is incomplete but representative:

- LOAD and STORE interact with external storage, respectively reading and writing relations.
- Standard embarrassingly parallel commands: FOREACH transforms every tuple in a relation and FILTER selects tuples from a relation based on a condition.
- Commands related to ordering and equality: ORDER BY performs sorting, RANK adds fields describing sort order but preserves the existing order, and DISTINCT removes duplicates.
- Grouping: GROUP a.k.a. COGROUP, which can be applied to more than one relation at a time.
- Joins: CROSS and JOIN can be used to respectively form the Cartesian product or any kind of inner or outer join [Gro+09; ISO92] of two or more relations.

Most commands can utilize *functions* to specify their exact effects. For example, FOREACH could be used as `FOREACH r GENERATE f(x)` where `r` is a relation, `f` a function, and `x` a field contained in the tuples of `r`. The result of the command is a relation containing 1-tuples whose values are given by the function `f` on the field `x` of each tuple in `r`. There are many built-in functions, including arithmetic operators as well as aggregating functions such as COUNT, which computes the number of tuples in the given relation.

Clearly, these operations by themselves are much more expressive than the MapReduce model, but Pig Latin can also be extended by users. While the command set cannot be changed without modifying Pig itself, new functions can easily be added. Furthermore, the flexibility of the data model means that all user-defined functions can be used in any function-using command without restriction, unlike e.g. in Hive where SELECT clauses only allow using scalar functions.

Pig has been widely adopted. In June 2009 at Yahoo!, 60% of *ad hoc* and 40% of production Hadoop MapReduce jobs came through Pig, and further increases in Pig usage were expected [Gat+09]. A cross-industry study performed in 2012 showed three out of seven analysed clusters having significant Pig usage, one of which was observed to have had over 50% of MapReduce jobs submitted via Pig [Che+12]. LinkedIn [LiI] uses Pig both for user-facing data set generation and for analytics [Aur+12]. The reported runtime increase when using Pig instead of hand-written MapReduce has ranged from a factor of 1.3 [Sha+12] to 1.5, but it has improved significantly over time and is likely to continue to do so [Gat+09]. This level of performance loss seems to be acceptable in practice: consider that Twitter was using 'almost exclusively' Pig for its analytics in 2011 [Lin+11].

3.2 Apache Hive

Apache Hive [Hiv; Thu+09; Thu+10a] is a data warehouse system built on top of Hadoop: essentially, it is a high-level interface to both MapReduce and the backend storage system, which is typically HDFS, but can also be HBase (see Section 3.3). Hive enforces a structural view, very similar to traditional relational database systems [Cod70], of the data sets it handles. They are queried and manipulated using a language based on SQL (Structured Query Language) [Cha+74; Gro+09; ISO92] called *HiveQL*, which is translated to MapReduce computations. Hive was originally developed by Facebook; later, Google created a very similar warehousing solution called Tenzing [Cha+11]—a rare example of outside ideas being incorporated so directly at Google, instead of the other way around.

Hive's data model is based on *tables*, akin to those used in relational databases. Records of data are stored in *rows*, which are split among a set of typed *columns*, which are in turn defined in a *schema*. A row may have a null value in any column, but each row in a table always has the same amount of columns. Possible column types include primitive types such as integers and strings as well as complex types: arrays, key-value mappings, and product and sum types called *structs* and *unions*.

All metadata about the tables managed by Hive is catalogued in the *metastore*. The existence of the metastore, i.e. keeping track of persistent metadata about data sets, is what makes Hive a data warehouse system as opposed to purely computational systems such as Pig. The metastore remembers all tables and all information about them; primarily their schemata. Because it is randomly accessed, the metastore is not stored in HDFS. Instead, a traditional relational database is used.

Various settings for performance tuning may be applied to tables in Hive. Tables can be *partitioned* on certain columns, so that rows with the same combination of the partitioned columns' values are stored together. Partitions may furthermore be *bucketed*, which is another layer of partitioning based on the hash of a single column. Table rows can also be stored in sorted order. When using HDFS storage, tables map directly to directories, partitions to subdirectories of their table's directory, and buckets to files in their partitions' directories.

Notably, even though Hive manages storage of tables, it does not rely on any particular file format. As long as the contents of each file can be serialized for storage and deserialized (using a Java class called a *SerDe*) for manipulation in HiveQL according to the table's schema, the files comprising the data of one table can even be in completely different storage formats.

Hive supports *indexing* on table columns, a classical strategy for speeding up query operations in databases. The trade-off is that the index takes up some additional storage space and modifications become slower as the index needs to be updated. Considering that Hive's main use case, data warehousing, consists of managing very large and mostly immutable data sets, the slowdown is irrelevant and the amount of space taken by the index is likely to be negligible, whereas the query speedup is likely to be very welcome. Hive currently provides two kinds of indices: one that identifies HDFS blocks for the rows corresponding to a given key, and a bitmap index [Cha+98] that also identifies which rows in the blocks are populated with that key.

HiveQL currently (as of version 0.12) has two kinds of data manipulation statements: LOAD, which simply copies data files into the appropriate HDFS directory of the table, and INSERT, which writes the results of a SELECT clause into a table while performing appropriate format conversions. LOAD is an optimization, relying on the user to make sure that the file is usable in the table as-is, lest the table end up in an unusable state. INSERT is more flexible, as it can insert into more than one table at once and compute the partitioning dynamically. There are no other manipulation statements: HiveQL currently has no way of updating or deleting rows. This makes sense, given that rows are typically contained as-is in files in HDFS, which does not support in-place modification of files.

Querying in HiveQL is done with the SELECT statement, like in SQL. Various clauses to modify the statement's behaviour are supported, as in any modern SQL system. The following is a sample of what is available:

- WHERE selects only rows for which a given condition is true.

- `DISTINCT` removes duplicates from the result.
- `GROUP BY` groups data by the given columns' values.
- Sorting clauses: `ORDER BY` and `SORT BY`, the latter of which only guarantees sorting the output of each reduce task, thereby forming a partially ordered result. `ORDER BY` performs a global sort, but one must use the new Hive 0.12, or a later version, to avoid a poor implementation in which all data is sent to a single reduce task for sorting [HIV10].
- Combining the results of many selections in one query with `UNION ALL`.
- Joins: the various forms of `JOIN` can compute any form of inner or outer join [Gro+09; ISO92] of two or more tables, as well as the Cartesian product.

All in all the functionality available is very similar to that offered by Pig Latin, though HiveQL is not quite as flexible due to Hive's stricter data model. Nevertheless, just like Pig Latin, HiveQL can also be extended by users via user-defined functions. Hive users can define three kinds of functions: ordinary ones, which simply transform one row into another and are therefore always run within map tasks; *table-generating functions*, which can transform one row into multiple rows; and *aggregation functions*, which can combine multiple rows together and thus are run in reduce tasks, or in map tasks as part of a *Combine* function.

Hive also has support for creating *views* based on `SELECT` queries. Views are essentially named queries that are saved in the metastore, which can themselves be queried just like tables can. Conceptually, when a view is queried, the result of the view's defining query is computed, and then the original query is evaluated on that result. In practice, the two queries may first be combined into a single one which is executed directly on the tables used.

Hive has seen wide adoption. As the originator of Hive, Facebook has naturally been a heavy user, with over 20 000 tables and several petabytes of data in a Hive cluster in 2010 [Thu+10b]. As of 2013, their data warehouse, which likely continues to be largely Hive-based, has grown to over 300 PB [Pre; Tra13]. LinkedIn uses primarily Hive and Pig for its internal analytics [Aur+12]. A cross-industry study performed in 2012 showed four out of seven analysed Hadoop clusters having significant Hive usage, three of which had 50% of their MapReduce jobs, sampled over time periods ranging from days to months, submitted via Hive [Che+12].

As Hive is used especially for analytics, the fact that it makes use of the purely throughput-optimized MapReduce as a computational backend has been considered problematic. In an interactive setting the startup costs of a Hadoop MapReduce job are not necessarily insignificant, as they can even dominate the execution time of short computations [Pav+09]. Frameworks that attempt to solve this problem are presented in Chapter 5.

3.3 Apache HBase

Apache HBase [HBa] is an open source distributed data storage system based on the design of Google's Bigtable [Cha+06], enabling random read-write access to individual records in Big Data sets. This is a key advantage over HDFS or MapReduce, which only provide streaming access. In addition, as bulk operations on HBase tables can be performed using Hadoop MapReduce, no functionality is lost by relying on HBase instead of HDFS for data storage—though performance is of course lower than when using HDFS directly. HBase was originally conceived by Powerset as a foundation for their natural language search engine [Geo11]; though the engine never materialized (because Powerset was acquired by Microsoft before the engine was completed), HBase continues to be developed under the Apache Software Foundation.

HBase provides sorted three-dimensional lookup tables in a manner similar to traditional relational database engines, but with a much simpler data model, namely:

$$(\mathit{row} : \text{string}, \mathit{column} : \text{string}, \mathit{version} : \text{int64}) \rightarrow \text{string}$$

In other words, each data value, or *cell*, in a *table* is uniquely identified by a *row*, *column*, and *version*, of which the rows, columns, and values are simply arbitrary byte strings while versions are 64-bit integers—typically timestamps. Data is sorted first by row, then by column, and finally by version, with later versions coming first in the sort order. This simple model allows scaling by just adding more nodes, without having to worry about maintaining the complex invariants to which relational databases adhere [HBR; Whi09].

HBase has a very simple interface to tables, consisting of only four operations (excluding metadata-related functionality):

1. *Get*: reads a row, possibly limiting the result set further to specific columns and/or versions.

2. *Put*: writes a row, either adding a new one or replacing an existing one.

3. *Delete*: removes a row.

4. *Scan*: iterates over a sequential range of rows, returning one at a time to the user.

This limited set of functionality makes HBase's essential nature as a key-value store evident: HBase itself does not provide the more complicated operations that are typically found in database systems, such as joins. As previously mentioned, however, Hive can use HBase as a storage backend, allowing that kind of functionality to be used on data stored in HBase.

As MapReduce handles scheduling computations on a distributed system, so does HBase take care of distributing the data it stores among the available nodes. Tables in HBase are automatically partitioned into sequences of rows called *regions*, which can be distributed among the HBase servers, aptly called *regionservers*. This spreads out computational load on the table as well as the data itself, enabling large tables to utilize the entire cluster's storage space.

HBase naturally also includes fault tolerance, which is mostly reliant on a reliable storage system, typically provided by HDFS. As with MapReduce and HDFS, it is based on a master-slave architecture where the master only co-ordinates the slaves and monitors their health. The slaves in an HBase cluster are the aforementioned regionservers. Unlike MapReduce and HDFS, HBase provides fault tolerance for the master node: this is facilitated by using Apache ZooKeeper [Zoo], a co-ordination service based on the Zab algorithm [Jun+11] (similar but not identical to the classical Paxos algorithm [Lam98]). ZooKeeper is used to make sure that only one master is active at any given time, and also to store various metadata about the cluster.

Fault tolerance on the regionservers requires some work due to the method used to implement write operations. For performance, writes (including additions, modifications, and deletions) are performed on in-memory caches called *MemStores* (in Bigtable, *memtables*) and only flushed periodically, to HDFS files called *StoreFiles* or *HFiles* (corresponding to the Bigtable *SSTables*, short for Sorted String Tables [McK+09]). Data loss is prevented by also logging writes to HDFS: when a regionserver fails, its log is replayed by all replacement regionservers (i.e. all servers that are assigned any region that was previously assigned to the failed server), bringing them up to date. Note, however, that currently (as of version 0.96) HBase does not ensure that log entries are flushed to physical disks before proceeding

with the operation [HBA12; Hof13]: therefore, in the event of power loss or a similar catastrophic failure, data loss can still occur.

Recall that HDFS does not allow modifying files. Thus, whenever a regionserver decides to flush a MemStore to HDFS, it creates a new StoreFile for the contents of the cache. Read operations must, in the worst case, consult the MemStore as well as all StoreFiles. As data is kept in sorted order, e.g. reads requesting only the latest version of a record might need to consult only the MemStore, but in the worst case, a read operation involves traversing the whole MemStore as well as all StoreFiles before the appropriate values to return are found. To prevent having to consult too many StoreFiles, they are periodically merged into a single StoreFile in a process called *major compaction*. At this point, all deletions are also fully handled: when a cell that is not currently in the MemStore is deleted, the delete operation is merely noted in a marker called a *tombstone* and eventually flushed, but the supposedly deleted cell still persists in the older StoreFiles. The cell and the corresponding tombstone are actually removed from storage only during a major compaction, when they are discarded from the final, merged StoreFile. *Minor compactions*, in which only a subset of the StoreFiles are merged and deletions are not processed, also occur occasionally.

Figure 3.1 provides a graphical overview of how operations in HBase affect the different kinds of state. In summary:

1. Write operations, including additions, modifications, and deletions, are logged and then applied to the MemStore.
2. The MemStore is eventually flushed, creating a new StoreFile.
3. StoreFiles are eventually merged together into a single StoreFile during a minor or major compaction.
4. Read operations access all StoreFiles and the MemStore.

Since StoreFiles are written only when flushing or compacting, the amount of records written at a time is typically quite large. Therefore compression can be utilized more effectively than in systems that simply modify or append to existing files: each StoreFile can be compressed as a whole at its creation time, resulting in a better compression ratio than could otherwise be achieved. Additionally, as major compactions are usually run when the HBase cluster is not under heavy load, it is possible to apply a relatively resource-intensive but effective compression algorithm on a large amount of data at once, improving compression ratios even further. (For some information about compression, see Chapter 6.)

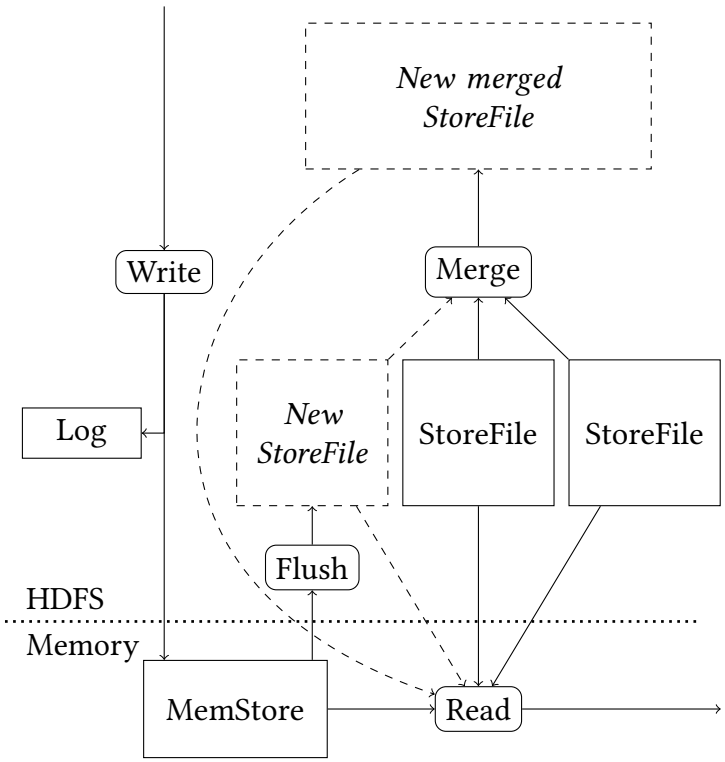

Figure 3.1: HBase state and operations. 'Read' includes both single-row reads and scans and 'Write' includes single-row additions or modifications as well as deletions. The boundary between HDFS and the MemStore is shown as a dotted line.

Having to read from several HDFS files for every read operation would be prohibitively slow. Hence, to speed up reads, regionservers cache parts of StoreFiles as well as individual lookup results, and allow using Bloom filters [Blo70] to quickly exclude StoreFiles from being considered for a query. Bigtable tests in Chang et al. [Cha+06] show that despite these efforts, random access reads were approximately an order of magnitude slower than similarly random writes, and sequential reads were either significantly slower or faster than sequential writes. Results from the Yahoo! Cloud Serving Benchmark [Coo+10] in 2010 demonstrated similar behaviour in HBase: while it dominated the competition in write-heavy workloads, HBase was comparatively slow in performing read operations.

Facebook has used HBase heavily with positive results: in 2011, Facebook's HBase clusters consisted of thousands of nodes implementing different kinds of applications, including real-time messaging among millions of users [Aiy+12; Bor+11]. Several other industrial users of HBase exist [HBP],

but none have (or have published information about) notably large cluster sizes or data volumes.

Chapter 4

Hadoop in bioinformatics

> In 26 years of software engineering, I have never come [across] a problem domain that I found stable enough to trust.
>
> Robert C. Martin [Mar96]

The field of bioinformatics contains a large number of Big Data problems, especially in sequencing data analysis. The tools offered in the Hadoop project have been heavily used in implementing various solutions, although other systems—mainly the Message Passing Interface, MPI [MPI93]—have been the method of choice for some projects [Tay10].

A task that has seen a significant amount of attention is *sequence alignment* or *mapping*: similarity search between two or more sequences in order to estimate either the function or genomic location of the query sequence. Alignment is an important part of almost any analysis process. As such, it is not surprising that much effort has been spent in developing efficient and scalable alignment methods.

CloudBurst [Sch09] and CloudAligner [Ngu+11] are examples of sequence aligners based on Hadoop MapReduce. CloudAligner is notable in that it uses map-only jobs to achieve greater performance. The publication that presented the Hadoop-based CloudBLAST [Mat+08] compared it against a similar MPI implementation, mpiBlast [Dar+03], finding that CloudBLAST performed up to approximately 30% better and was simpler to develop and maintain. Many MPI-based aligners [dAra+11; Mon+13; Rez+06] have nevertheless been created.

Alignment tools often include other features, either as additional utilities or because they are intended for some specific analysis for which alignment

is only a part of the process. The following are all examples that use Hadoop MapReduce for scalability. Seal [Pir+11a; Pir+11b] provides an aligner which includes postprocessing, such as duplicate read removal. Crossbow [Lan+09], Myrna [Lan+10], and SeqInCloud [Moh+13] implement sequence alignment as part of their specific analysis pipelines.

Sequence alignment is, of course, not the only analysis task in bioinformatics for which Hadoop has been utilized. The SeqWare Query Engine [OCo+10] uses HBase to implement a database for storing sequence data. MR-Tandem [Pra+12] carries out protein identification in sequence data using MapReduce. CloudBrush [Cha+12] and Contrail [Sch+] use MapReduce in performing a process called *de novo assembly*: assembly of previously unknown genomes from sequence data. SAMQA [Rob+11] detects metadata errors in sequence data files, using MapReduce for parallelization.

Finally, some projects provide support facilities, making it easier for their users to implement the complete analysis pipelines. The Genome Analysis Toolkit (GATK) [McK+10] is one example. It is based on the MapReduce model but does not use Hadoop, instead running on a custom engine and having a separate wrapper for distributed computing called GATK-Queue [GAQ]. The aforementioned Seal project, while focused on alignment, presents its functionality as a set of tools that can be used for other purposes as well. Cloudgene [Sch+12] is a platform providing a graphical user interface for executing bioinformatics applications based on Hadoop MapReduce, with support for several of the tools mentioned here. BioPig [Nor+13] is a Pig-based framework containing various useful functions, including wrappers for some other commonly used applications.

The author and his colleagues have developed two supporting tool sets of their own, offering useful functionality that was not previously available. Hadoop-BAM is a library providing file format support along with some useful command-line tools, and SeqPig is a higher-level interface in Pig including special functionality for sequence data analysis. They are presented in the following two Sections.

4.1 Hadoop-BAM

Hadoop-BAM [HBM; Nie+12; Nie11] is a library written in the Java programming language, providing support for using Hadoop MapReduce to manipulate sequencing data in various common file formats. Currently, as of version 6.0, the formats supported are all of the following:

- Sequence Alignment/Map or *SAM* as well as its binary representation, Binary Alignment/Map or *BAM* [Li+09; SAM13]. Originally only BAM was supported, giving Hadoop-BAM its name.
- Variant Call Format or VCF and its binary representation, Binary Call Format or BCF [BCF; Dan+11].
- The format originally created for the FASTA set of tools [Pea+88], which is nowadays known as the 'FASTA format' or simply FASTA.
- FASTQ [Coc+10], a simple extension to the FASTA format.
- QSEQ [CAS11], a file format that is output directly by some sequencing instruments.

Hadoop-BAM has both input and output support for all the above formats apart from FASTA, which can only be input. BAM and BCF are discussed further in Section 6.1.

Command line tools for some tasks commonly performed on SAM and BAM files are also included in Hadoop-BAM, with inspiration from the SAMtools [Li+09] software package. One such tool can sort and merge SAM and BAM files using Hadoop MapReduce, which is an important preprocessing step e.g. for visualization [Pab+13] and can benefit greatly from the parallelization of MapReduce. Testing it on a 50.7 GiB BAM file, near-linear scaling has been observed when using a Hadoop cluster with up to eight slave nodes: see Figure 4.1. The reduced speedup thereafter can be attributed to the relatively small file size leading to quite little data being allocated to each worker node. The machines used in this experiment were the same as those used for the experiments described in Chapter 7—their relatively low number of disk drives also explains why the scaling was fairly limited overall, as sorting is a very I/O intensive operation. Significant comparisons to other software were not performed, as none implement sorting BAM files in HDFS. However, as a simple baseline, the single-threaded `sort` command of SAMtools was tested; operating on local disk on the same hardware, it was over twice as slow as the single-slave Hadoop MapReduce job.

BAM input support in tools is a common desire among bioinformaticians, but this desire is often left unfulfilled due to the complexity of the BAM format. Hadoop-BAM is thus often used mainly for its BAM-related functionality. The Seal project donated FASTQ and QSEQ format support to Hadoop-BAM, and later began using Hadoop-BAM for SAM and BAM as well. SeqInCloud's genome analysis pipeline incorporates Hadoop-BAM

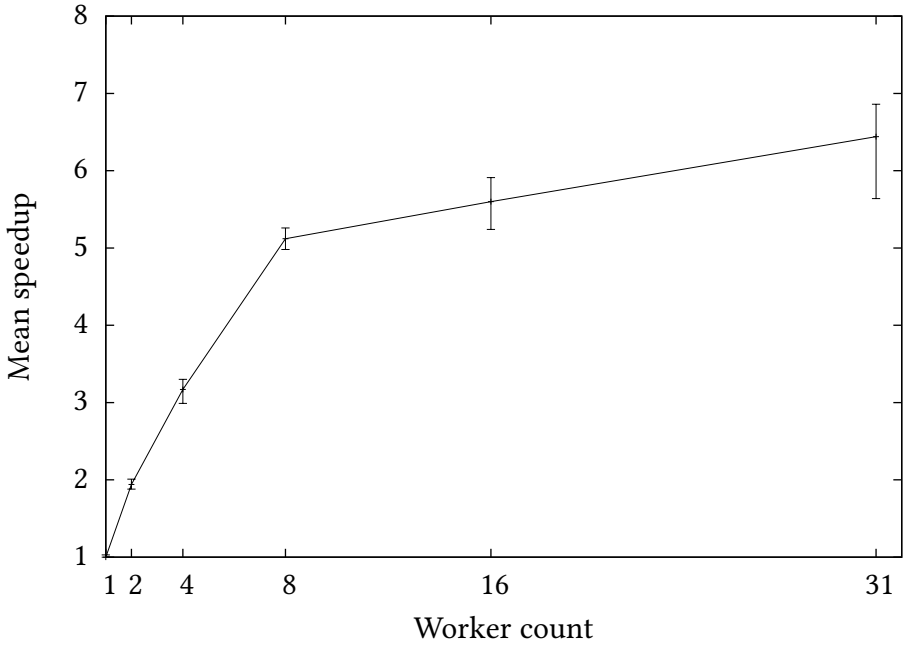

Figure 4.1: The speedup observed when sorting a 50.7 GiB BAM file with Hadoop-BAM. Mean, minimum, and maximum speedups for each worker count are indicated.

for BAM input. SAMQA relies on Hadoop-BAM for reading both SAM and BAM. Cloudgene contains Hadoop-BAM's sorting tool among its set of supported applications. ADAM [Mas] has used Hadoop-BAM to convert FASTA, SAM, and BAM files to the Parquet [Par] format, which has been designed for efficient processing in Hadoop.

4.2 SeqPig

While Hadoop-BAM gives developers the opportunity to create custom Hadoop MapReduce applications for sequencing data with control over every aspect of processing, SeqPig [Sch+13a; Sch+13b; Seq] is a high-level interface based on Pig. With SeqPig, as long as the application can be adequately described in Pig Latin, development is simpler and does not require familiarity with MapReduce or Java.

The latest version of SeqPig, 0.5, provides almost the same file format functionality as current Hadoop-BAM, lacking only the recently implemented VCF and BCF: SAM and BAM, FASTA (read-only), FASTQ, and QSEQ

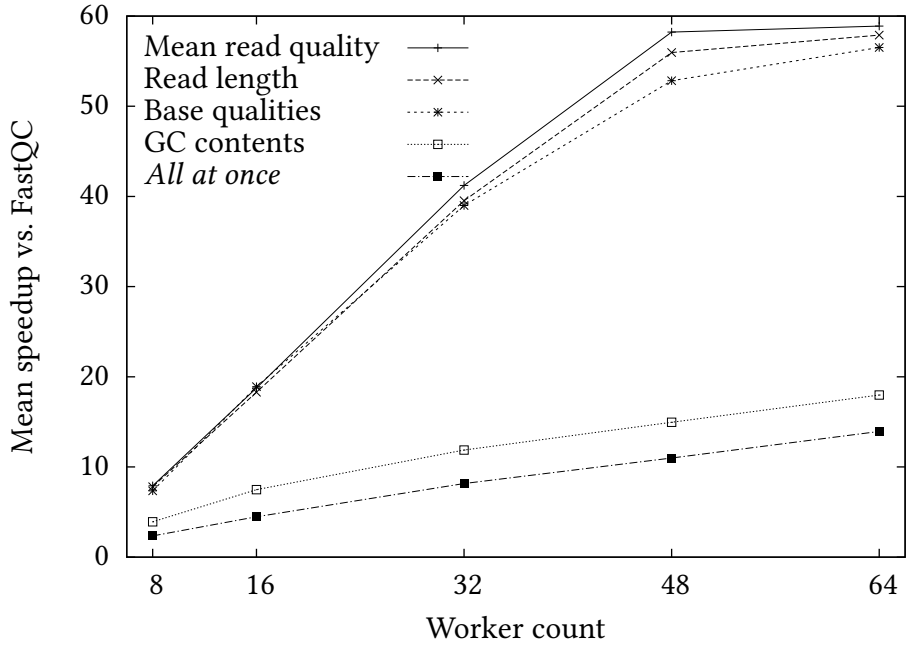

Figure 4.2: The mean speedup of SeqPig vs. FastQC in computing various statistics over a 61.4 GiB FASTQ file. Note that the sets of statistics computed by the SeqPig script and FastQC are similar but not identical.

are supported. All data and metadata in these formats can be loaded for manipulation in Pig Latin. In addition, SeqPig includes user-defined functions for several useful operations specific to sequencing data. Thanks to Pig, all processing can take place scalably using Hadoop MapReduce. Figure 4.2 provides an example of such scalability, showing the speedup of computing certain read quality statistics in SeqPig compared to the single-threaded FastQC [And] tool. For details about this experiment, including the precise functionalities compared as well as the software and hardware configurations involved, the reader is referred to Schumacher et al. [Sch+13b].

The unrelated BioPig [Nor+13] project naturally shares the advantages of Pig with SeqPig. The differences between the two lie in their provided bioinformatics-specific functionality. In terms of file formats, BioPig supports only FASTA and FASTQ—although, unlike SeqPig, it has output support for FASTA. Otherwise, the sets of user-defined functions provided by SeqPig and BioPig are intended for very different concerns in sequencing data analysis. For this reason, one may wish to use SeqPig and BioPig together, and due to Pig's simple data model, this is highly straightforward.

Chapter 5

Interactivity

> Software is getting slower more rapidly than hardware becomes faster.
>
> 'A Plea for Lean Software'
> Niklaus Wirth [Wir95]

The Hadoop-based analysis frameworks that have seen the most use thus far, in bioinformatics as well as other fields, are Pig and Hive, both of which are based on Hadoop MapReduce. Unfortunately, MapReduce is optimized for throughput at the expense of latency, and is not suitable for interactive tasks [Pav+09]. In order to overcome the performance limitations inherent in MapReduce, other frameworks, specialized for *ad hoc* exploratory and interactive analysis, have been developed. Many of them are, like Hive, SQL-based query systems, possibly due to the influence of Google's Dremel [Mel+10]. As such, the remainder of this Thesis also concentrates on the SQL-based systems in order to make comparisons more meaningful.

The two best established freely available contenders at the moment are Shark [Xin+12] and Cloudera Impala [Imp]. Apache Drill [Dri] is another freely available effort, but is still in early stages of development. Both Drill and Impala are largely based on Dremel's design. Some proprietary systems also exist, including Amazon Redshift [Red] and HAWQ [HAW13]. Facebook's Presto [Nov13; Pre] was also included in that group, until it was made open source mere weeks before the completion of this Thesis [Tra13]. Due to Drill's lack of maturity, the inaccessibility of the proprietary systems, and the recentness of Presto's release, these others were not evaluated in this work.

BlinkDB [Aga+12; Aga+13; BDB] is yet another SQL-based interactive query system, but with a unique approach: it speeds up queries by running on only a subset of the full data sets involved, and computes an upper bound on the error in the result. Users may perform either error-bounded queries, using a relative error coupled with a confidence interval, or time-bounded queries, where the most accurate answer that can be computed in a given time limit is returned, along with an estimate of the error at a certain confidence. BlinkDB executes its queries with either Hive or Shark. Like Presto, BlinkDB was released when this Thesis was already nearing completion, and therefore was also not evaluated in this work.

Apache Tez [Mur+13; TeH; Tez], a part of Hortonworks's Stinger Initiative [StI], is a computational framework designed with interactive query tasks in mind. Versions of Hive and Pig that can use Tez instead of MapReduce are in development, and are expected to demonstrate improved performance compared to MapReduce. A Tez job consists of an arbitrarily large directed acyclic graph of tasks, avoiding various intermediate communication requirements compared to an equivalent set of MapReduce jobs, such as having to flush each job's output to disk and having to wait for the previous job to complete. As Tez is in early development stages, it is not evaluated in this Thesis.

Shark and Impala are both data warehouse systems similar to Hive. In fact, they are both compatible with Hive, in that they use the same metastore system and thus operate on tables in exactly the same way as Hive. Their query languages are also very similar to Hive's HiveQL, with the main differences being that Hive tends to support some operations that Shark and Impala do not [ImU; ShC].

The rest of this Chapter concerns the inner workings of Shark and Impala. While Impala implements its own computational engine, Shark is based on a system called Spark, which is delved into before considering Shark-specific matters.

5.1 Apache Spark

Apache Spark [Spa; Zah+12], developed at the UC Berkeley AMPLab (the Algorithms, Machines, and People Lab of the University of California, Berkeley), is a distributed computing framework similar to e.g. Hadoop MapReduce, but based on a substantially different model. The main motivation of Spark was to improve the performance of two classes of tasks. The first was interactivity, the main focus of this Chapter. The second was iterative algorithms—and more generally, any task in which re-use of in-

termediate results is key. MapReduce is not a good fit for such algorithms due to its rigid single-pass system: each iteration of a loop, for example, would have to be a separate MapReduce job, and the only way in which later iterations can use the output of earlier iterations is by having the data written to a shared storage system, such as HDFS. This is, of course, an excessive use of resources compared to keeping data in the local memory of each node and performing all processing therein. Iterative systems such as HaLoop [Bu+10; HaL] and Pregel [Mal+10] (and its open source counterpart, Apache Giraph [Gir]) solve this problem for certain kinds of computations, but Spark provides a general-purpose abstraction for distributed in-memory computing.

The abstraction Spark is based on is the *RDD* (Resilient Distributed Dataset) data structure. An RDD is a read-only set of *partitions* containing records, created by any number of *transformations* on an originating data set. Note that the amount of transformations may be zero: this way the data sets themselves are also RDDs, with the partitioning typically being a natural consequence of the storage system—e.g. treating each HDFS block as one partition. In derived RDDs, the records in a partition are not necessarily stored at any given time. Instead, their *lineage*, the sequence of transformations needed to compute them, is always known, allowing missing partitions to be computed on demand. This mimics the *lazy evaluation* [Fri+76; Hen+76] or *call-by-need* [Wad71] strategy found in some programming languages.

Transformations in Spark are similar to the *Map* and *Reduce* functions of MapReduce: side effect free higher-order functions which are applied in parallel to the entirety of the data. Compared to the fine-grained interface of most other in-memory frameworks such as traditional distributed shared memory systems [Nit+91] or HBase [HBa], operations like these have an advantage in that fault tolerance can be provided very cheaply by knowing what computations were to be performed by failed nodes and re-executing them as needed, akin to MapReduce. More expensive methods such as logging each record update separately or replicating the output of each intermediate stage are not needed. The fact that transformations in Spark are side effect free also allows diminishing the effect of stragglers via speculative execution, as in MapReduce. The following are examples of transformations, demonstrating the variety of operations available:

- *map* and *filter* perform the usual function application and predicate-based selection tasks.
- *flatMap* is akin to MapReduce's *Map* in that the mapping function can emit any number of outputs for one input record.

- Set operations: *union* and *subtract*.
- Joins of two RDDs of key-value pairs: *cartesian* computes the Cartesian product while *join* performs a hash join [DeW+84]. Left and right outer joins [Gro+09; ISO92], are also available.

Each partition in an RDD has a (possibly empty) set of partitions it depends on. Contingent on what information is available to Spark, this set of dependencies may or may not be minimal. For example, if the parent RDDs are partitioned by hashing, then each partition in the result of a hash join (the *join* transformation) depends on only one partition in each parent: the partitions with the set of hashes that are assigned to that output partition. If the partitioning is not known, then each partition in a join's RDD must depend on all partitions in each parent RDD.

Spark can be told to *persist* RDDs: a hint that the RDD is likely to be re-used and thus its partitions should be held in main memory—or, if there is not enough room, on local disk storage, instead of being recomputed when needed. Whether an RDD is persisted or not comes into play when a new partition that does not fit in memory is computed. To make room for the new partition, a partition from the least recently accessed RDD is evicted from memory. If that RDD was persisted, the old partition will be saved to disk, otherwise it will simply be deleted. As an exception to this scheme, partitions from the same RDD as the new partition are not evicted, because they are likely to be needed soon. Persistence only to disk or replication among multiple nodes can also be requested. In addition, *serialization* can be controlled: by default, partitions are stored deserialized, as Java objects, and serialized only when they are moved to disk. The high memory overhead of Java objects [Bac+02; Xin+12] means that in certain cases it is possible to gain performance by deserializing only on demand.

To actually retrieve or store data from RDDs, *actions* are used in a driver program or interactive shell. The following are examples of common actions:

- *collect* returns the records in the RDD, storing them in a list in the calling program.
- *count* returns the number of records in the RDD.
- *reduce* folds the RDD to a single value by applying a given commutative and associative function to the records.

Until an action is used, no work is performed on the cluster. This allows optimizing the execution plan as a whole. Because of the fine-grained

dependency tracking, needed but missing partitions do not necessarily require computing the preceding RDDs fully. Thus a large part of the intermediate results can be re-used, as long as they are available due to either persistence or simple co-incidence.

Spark is implemented in the Scala programming language [Ode+06], which is concise enough to allow convenient interactive use of the Spark API (application programming interface) while making it possible to use APIs written in Java, such as that of HDFS, directly. Like most distributed systems presented in this Thesis, Spark's architecture consists of a single master, which monitors the health of the cluster and schedules jobs, and several slaves which carry out the computations themselves.

Altogether, due to the relatively high-level implementation language that can also be used interactively (as an alternative, there are bindings to the Python programming language [Pyt] as well) combined with the wide array of available transformations and actions, the capabilities of Spark are actually closer to Pig than to e.g. Hadoop MapReduce. Spark is significantly more performant, however: while users of Pig have reported best-case performance of approximately 0.74 times that of equivalent hand-written Hadoop MapReduce code [Sha+12], testing Spark showed that performance ranged from about twice that of Hadoop MapReduce up to an about 40-fold speedup, depending on the application [Zah+12].

5.2 Shark

Shark [Sha; Xin+12] is a data warehouse system built on top of Spark, with an additional focus on running queries fast enough for interactive use. Just like Hive, Shark treats data sets as tables and allows querying and manipulating them with HiveQL. While Hive translates HiveQL statements to Hadoop MapReduce jobs, Shark executes them with Spark instead. Shark can read and update Hive's metastore, and thereby is capable of being a full replacement for Hive. Some less commonly used Hive features are not yet supported, however [ShC].

Like Spark, Shark was developed at the UC Berkeley AMPLab and is written in the Scala programming language. Being written in Scala, it can utilize the Hive API directly and so is compatible with user-defined functions and file format functionality written for Hive, including user-defined table-generating and aggregation functions. Such a level of compatibility is naturally important for making Shark a drop-in replacement for Hive. Notably, Shark can be used as an API in Spark-using programs, allowing HiveQL queries to be translated into corresponding RDDs. Therefore it is

relatively simple to use Spark for operations that are cumbersome to express in HiveQL, all the while performing I/O on tables in a Hive-compatible way and having Spark optimize the entire computation as a whole.

In order to efficiently run HiveQL queries, Shark implements additional features on top of the RDD model of Spark. Instead of using Spark's standard in-memory storage layout, in which individual records are simply stored as lists, Shark uses a *columnar* or *column-major* storage layout: each column of a table is stored in a single array. Thus e.g. the first record in the table corresponds to the collection of the first elements of each such array. This reduces Java object overhead compared to Spark's deserialized in-memory storage, and allows for further memory savings via compression: a compression scheme appropriate for the column's data type can be used in each column, with negligible processing cost compared to not compressing at all. (For more details about columnar storage, see Chapter 6.)

Aside from columnar in-memory storage, Shark's main improvements on top of Spark involve making decisions based on statistics observed in the data. Whenever data is loaded into memory, per-column statistics are gathered in order to make better decisions concerning the columns' in-memory compression schemes. For example, if a column contains only few distinct values, simple lookup table encoding is used. Statistical guidance is also applied at any point in the execution plan when output partitions can be affected by multiple input partitions, i.e. the next RDD is defined by an operation such as *reduceByKey* or *join* instead of one-to-one operations like *map* or *filter*. The workers outputting the input RDDs' partitions compute the distribution of the incoming data, and based on that the join strategy and degree of parallelism can be affected. This is termed *partial DAG execution*, referring to the execution plan as a directed acyclic graph or DAG.

Fair warning: no released versions of Shark implement partial DAG execution. An implementation can be found in a development branch based on the fairly old 0.2 version of Shark [SPD]. Porting the code to a more recent release is intended [SHA12], but as of Shark 0.8 this has not been worked on.

The Shark developers, have together with users with whom they have worked, reported speedup factors of up to 100 compared to Hive, with several queries that took minutes to complete in Hive being completed in less than a second on Shark [Xin+12]. Neither Shark nor Spark have seen widespread adoption as of yet, though.

5.3 Cloudera Impala

Like Shark, Cloudera Impala [ImD; ImG; Imp; Kor+12; Leb13] is a Hive-compatible data warehouse system using Hive's metastore and HiveQL. Unlike Hive and Shark, which perform their computations on MapReduce and Spark respectively, Impala uses its own computational backend based on the design of Dremel [Mel+10]. Impala is in an early stage of development: the first stable version was released as recently as May 2013. Dremel and Impala were never intended to replace MapReduce, but rather to complement it with interactive analysis capabilities. As such, Impala is meant to be used alongside Hive, and the fact that it lacks many of Hive's commonly used features [ImU] is not critical.

Impala is written mostly in the C++ programming language [Str13], a notable departure from the languages intended for the JVM (Java Virtual Machine) that are used by most Hadoop projects. This makes re-using Hive features in the Impala code base significantly more difficult, which explains why Impala's feature set is so limited compared to that of Hive or Shark. The situation is even worse when performance is desired, due to the overhead of using JVM code from native code [Daw+09]. Likely as a result of this, Impala currently (as of version 1.1.1) includes no form of user-defined function or file format support and so is completely inextensible by users.

Distinctly from all distributed systems examined in this Thesis thus far, in Impala all nodes are essentially equal: there is no static master-slave division. While there is a *statestore* process which monitors node status, its only purpose is to inform the Impala workers about nodes that have become unreachable: query execution does not directly involve the statestore at all, and the statestore may fail without preventing further queries from being executed. Since all nodes are equivalent to one another, queries can be submitted to any node for execution. That node then becomes the *co-ordinator* for that query: a sort of per-query master node. The co-ordinator constructs an execution plan for the query, and instructs the nodes that have the input data locally available (i.e. are storing blocks of the relevant HDFS files) to perform the necessary computations. These worker nodes stream results back to the co-ordinator over the network—never writing to disk—, with the co-ordinator performing any final reductions needed before displaying the end result to the user.

As streaming all intermediate data to one node could be prohibitively slow, Impala is capable of distributing the reduction phase by forming an *execution tree*. The leaf nodes in the tree are the nodes that have the data in local storage, the root node is the co-ordinator, and intermediate nodes reduce data on the way from the leaves to the root, streaming data both

in and out. Unlike Dremel, Impala currently (as of version 1.1.1) does not form arbitrarily deep execution trees [ImT13], which can limit performance for some queries.

Impala does not provide fault tolerance of any kind for queries: if a node fails during a query, the entire query has to be re-run. The streaming-based architecture of Impala does make fault tolerance somewhat more onerous than in MapReduce or Spark: as intermediate results are streamed from node to node, they are never saved anywhere, so losing an intermediate node before it has completed its work means that the whole subtree rooted at that node must be re-executed. Regardless, it is unfortunate that no fault tolerance mechanism has yet been implemented.

In an effort to speed up query execution, Impala utilizes the LLVM compiler suite [Lat+04; LLV] for runtime generation of specialized native code for each query [Li13]. This avoids various overheads that would result from having to re-interpret the query at every stage of execution. Tests by Cloudera show a near threefold speedup compared to not using code generation [Li13].

Chapter 6

Storage formats

> Clutter and confusion are failures of design, not attributes of information.
>
> *Envisioning Information*
> Edward R. Tufte [Tuf90]

The high-level manner in which data is accessed in tabular data warehousing systems such as Hive, Shark, and Impala leaves the choice of the actual storage format open. As long as the data files comprising a table can be interpreted as collections of records in accordance with the table's schema, the files could theoretically even each be stored in a different format—though the implementations are not all that flexible in practice.

Many distinct file formats are used to store sequencing data [FAQ], with the choice depending on the use case. They tend to be highly application-specific and thus very different to the generic formats supported by the data warehousing systems. In order for the data to be made available in tables, either it must be converted to a supported format or the necessary features for using it directly must be implemented in the warehousing system. Thanks to the extensibility of Hive and Shark, the latter option is possible without having to directly modify the systems themselves. Extending Impala with new file format support would be a far more arduous task. Overall this means that the option of using the original sequencing data formats exists, but only for Hive and Shark (and any other compatible systems). However, it would then be possible to use Hive or Shark to copy the data from a table backed by sequencing data formats to another with a more generic file format, thereby converting between the two and providing many more systems with access to the data.

Traditional text files, typically encoded as UTF-8 [Pik+93], are commonly used for their simplicity and readability, especially in the realm of sequencing data. Text files are attractive for their capacity to be manipulated without requiring specially purposed tools. Their downside is that they waste significant amounts of space compared to binary encodings, which of course also slows down their processing. For small files, such as typical BED (Browser Extensible Data) files [Qui+10], the overhead is usually insignificant, in which case textual storage is an acceptable option. Often the existing format can even be used directly as the table's data: HiveQL can define tables backed by text files where the separation between records and between fields within a record is encoded as a specific character. For example, BED files have one record per line and thus the record separator is a newline character, and their fields are separated by the horizontal tab character.

Binary storage formats, which do not restrict themselves to the readability of text files, are usually a better choice for large data volumes: they take less space than textual formats and can therefore also be processed faster. Some custom-designed binary formats are used for sequencing data, such as the BAM [Li+09] and BCF [Dan+11] files that are described in detail in Section 6.1. Except for their unique encoding methods, they are not very different to the more generic formats supported by the data warehousing systems, such as the SequenceFiles [SeF] used in Hadoop or the binary container format of the Apache Avro [Avr13] system.

Both textual and binary formats are usually losslessly compressed for further space savings, using a compression scheme with low enough processing costs so that the performance loss due to compression and decompression is insignificant compared to the gain due to having to perform less I/O operations. The optimal selection depends on the computer hardware and its usage patterns:

- *Snappy* [Sna] and *LZO* (Lempel-Ziv-Oberhumer) [Obe] are common choices that are sufficiently low-cost so as to be an improvement in almost all cases.

- `gzip` [Deu96b], based on the DEFLATE [Deu96a] algorithm, is more expensive but also much better compressing.

- Comparatively highly expensive compressors include `bzip2` [Sew], LZMA (Lempel-Ziv-Markov chain algorithm) [Pav13], `zpaq` [Mah], and the `gzip`-compatible Zopfli [Ala+13; Zop]. Long-term archival is an example of a use case in which the improved compression ratios

achieved by these algorithms justify their costs, but for most uses they are unacceptably slow.

All storage formats mentioned thus far, despite their many differences, share one fundamental aspect: they encode one datum at a time in the most obvious manner. Excluding metadata, each format can be considered as a list of n records r_i ($i \in [0, n)$), where each record is encoded in a way depending on the format f in question: $enc_f(r_i)^n$. This commonality means that they are all *row-major* or *row-oriented* formats, a.k.a. adhering to the *N-ary storage model*. Here 'row' is to be understood as a record, e.g. as those forming a table in a data warehousing system.

As can be expected, all storage formats are not row-major: *column-major* or *column-oriented* or simply *columnar* formats, a.k.a. those using the *decomposition storage model*, have also been in use for a long time [Cop+85; Wie+75]. Instead of storing complete records at a time, they store values from a column at a time. I.e. for records with k columns $r_i = (c_{1,i}, c_{2,i}, \dots, c_{k,i})$, columnar storage files consist essentially of a tuple of lists of single-column values $(enc_f(c_{1,i})^n, enc_f(c_{2,i})^n, \dots, enc_f(c_{k,i})^n)$ instead of a single list of records. Note that because the columns are stored separately, it is possible to use a different encoding for the data comprising each column: $(enc_f^1(c_{1,i})^n, enc_f^2(c_{2,i})^n, \dots, enc_f^k(c_{k,i})^n)$. This way encodings and compression schemes that are suited only to specific kinds of data can be used effectively. And since values within a column are likely to have less Shannon entropy [Sha48] than each row considered as a whole, even generic compression schemes tend to be more effective [Flo+11]. Another advantage of column-oriented storage, which only affects certain usage patterns but can be the greatest performance improvement compared to row-oriented storage, is that it is possible for a computation to completely avoid performing I/O on columns that it does not need. This feature makes columnar storage an extremely attractive option for tabular data warehousing systems, as the processing cost of a query is then dependent on the sum of the sizes of the columns used in the query instead of the size of the table as a whole. The corresponding disadvantage is that if many columns are used in a query, fetching all the columns of a record is slower than in a row-major format due to poor spatial locality.

In an effort to achieve the best of both worlds, *hybrid* formats, in which relatively small groups of records are stored in a column-oriented fashion, have been developed [Ail+01; Han+03]. These are most similar to columnar storage formats, but instead of laying out all n values from each column in one sequence, the list of records is partitioned into p groups, and the n_j records in each partition P_j are stored columnarly. Thus the entire format

is of the form $(P_1, P_2, \ldots, P_p)$, where:

$$P_j = \left(enc_f^1(c_{1,i})^{n_j}, enc_f^2(c_{2,i})^{n_j}, \ldots, enc_f^k(c_{k,i})^{n_j} \right).$$

With a suitable partition size, this scheme realizes the advantages of columnar formats while keeping the cost of fetching complete records low.

The next Section will discuss row-oriented binary formats with support among Hive, Shark, and Impala, namely SequenceFiles and the Avro format, in somewhat more detail before considering two specific formats related to sequencing data: BAM and BCF. The remaining Sections summarize some of the column-oriented formats, which are actually all hybrids, that are relevant to Hive, Shark, and Impala: RCFile, ORC, Trevni, and Parquet.

6.1 Row-oriented binary storage formats

As mentioned previously, row-oriented formats are not very different to one another. The main differences are in the encoding schemes: the various ways in which values are converted into the binary storage format. Sequencing data formats tend to avoid wasting bits by taking advantage of the fact that the domains of the values are known: for example, in BAM the nucleotide bases comprising sequence data are stored as four-bit integers (two in each byte) since there are only $2^4 = 16$ possible values. More generic formats typically do not allow for benefiting from this kind of domain knowledge, relying instead on the compression scheme to produce a sufficiently compact result.

SequenceFiles [SeF], being mainly used with Hadoop MapReduce, always store key-value pairs. Metadata at the beginning of the file—a file *header*—names the Java classes corresponding to the key and value types. Since any classes that can be read and written by Hadoop (those implementing the `Writable` [Wri] interface) can be used, this is extremely flexible and extensible, as both the types and their encodings depend on the classes' implementations. However, relying on Java classes prevents programs that are incompatible with Java from accessing arbitrary SequenceFiles.

Avro [Avr13] files include a schema in the header, which specifies the type of data contained in the file. Arbitrary composition of the available primitive and derived types is possible. This way practically any type of data can be encoded, but unlike in SequenceFiles, the binary encoding of each type has been defined ahead of time in the Avro specification. As such Avro files are more portable than SequenceFiles, but may not be as efficiently encoded. On the other hand it is possible to deserialize only

the needed parts of records in an Avro file, whereas in SequenceFiles each record can be decoded only as a whole.

Both SequenceFiles and Avro files contain *synchronization markers*, which are important for allowing efficient parallelization when operating on large amounts of data. The distributed computing engines mainly discussed in this Thesis—Hadoop MapReduce, Spark, and Impala—all conceptually split files into parts, assigning different non-overlapping ranges of the input data files to different computational tasks. Because the number of parts is not known when a file is written (and can of course be different every time the file is used), the offsets within the file demarcating the splits are computed dynamically, typically based only on the file size and the number of parts. The problem with this is that it can result in a data record being split along the middle. Thus synchronization markers are placed between records, or relatively short sequences of records, so that it is possible for readers to *synchronize* to record boundaries by scanning for a synchronization marker before beginning to read data. Both SequenceFiles and Avro files use a random 128-bit synchronization marker which is given in the file header. The length and randomness greatly reduce the chance of a collision with any input data.

The following two Sections discuss compression and BAM and BCF respectively. In particular, performing synchronization in the presence of compression and with file formats that lack synchronization markers, like all sequencing data formats, is explained.

Compression schemes

Compressors ordinarily work on the entire sequence of input data they are given, producing a single compressed output sequence that can be decompressed back to the original. For typical uses, there are no issues with compressing an entire data file in this manner. But when dealing with Big Data, even individual files can be so large that computations on them should be parallelized: file splitting is necessary for good performance. Compressing a file as a whole makes it impossible to access data in the middle of the file without decompressing everything from the start of the file, so splitting a compressed stream for parallel access is essentially useless. Hence, to be useful in a distributed computing context, the file format should wrap the underlying compressor in such a way that parallel access is possible.

Many compressors do have a relatively small maximum *block size*: they only compress at most a certain amount of input data at a time, emitting several compressed blocks for one input data sequence. This limits the max-

imum memory usage of both the compressor and the decompressor, and for some algorithms, such as the Burrows-Wheeler transform [Bur+94] used in `bzip2`, is the only way of making the compressor and decompressor work incrementally. But, despite emitting blocks that could be logically assigned to separate tasks, compressors do not tend to include synchronization markers, so reliable splitting is not possible. Furthermore, since most compressors are unaware of the format of the data they compress, it is possible for records in the underlying data to be split up across block boundaries. As long as synchronizing to the record boundaries after decompression is possible, this is not a problem, but it makes correct splitting more difficult and means that tasks may have to decompress an extra block beyond their assigned range—a minor performance loss.

SequenceFiles and Avro files both allow compressing sequences of records contained within. This is termed *block compression*. Note that the term 'block' here refers to consecutive sequences of records, not the previously discussed compressor output blocks: a single file-level 'block' may consist of any number of compressed blocks, depending on the compressor used. Since the synchronization markers are not part of the records themselves, they are not compressed and thus synchronization can be performed as usual while having compressed the bulk of the data.

SequenceFiles additionally support *record compression*, where only the value in each key-value pair is compressed. The overheads of most compression formats typically cause this form of compression to be a net loss unless the values are exceptionally large.

BAM and BCF

The BAM (Binary Alignment/Map) [Li+09; SAM13] file format is a row-oriented binary format consisting of a custom encoding of the textual SAM (Sequence Alignment/Map) [Li+09; SAM13] format. BAM is additionally always compressed using the BGZF ('Blocked GNU Zip Format' according to e.g. Cánovas and Moffat [Cán+13] and Cock [Coc11]) compressor. BCF (Binary Call Format) [BCF; Dan+11] is to VCF (Variant Call Format) [Dan+11] as BAM is to SAM. BCF may also be BGZF-compressed, but unlike with BAM, compression is not mandatory.

Neither BAM, BCF, nor BGZF contain synchronization markers. In addition, while BGZF is explicitly block-based and was originally introduced as part of BAM, records are allowed to cross BGZF block boundaries. All this makes correctly splitting BAM and BCF somewhat nontrivial.

Note that BAM and BCF can be, and commonly are, indexed. As the index contains the precise positions of many records in the file, one might

think that it could be used for aligning splits accurately. Unfortunately their indexing schemes are not suitable for the task, due to at least all of the following reasons:

- Indexing requires that the data be sorted by the co-ordinate field, because the intended use case is looking up records by co-ordinate ranges. If such an index were the only method of splitting, many files could not be split at all. Notably, sorting BAM and BCF is a commonly performed operation that can be significantly sped up with distributed computing, but it would be impossible if only sorted inputs were usable.

- The index is optional: it is not necessarily present even for appropriately sorted data. Thus it cannot be relied upon.

- The indexing scheme, based on the strategy used for the Human Genome Browser of the University of California, Santa Cruz [Ken+02] and closely resembling the layout of an R-tree [Gut84], is based on storing the file positions of certain predefined sequence co-ordinate ranges, called *bins*. The amount of records in a bin does not affect the index in any way and therefore, depending on the distribution of the data, even the smallest bins in terms of sequence length may be excessively large in terms of actual data length, i.e. as splits. In theory, an arbitrarily large file whose records all fall in the same bin is possible, in which case the index would effectively contain only one split.

- The encoding of the index is limited to sequences whose length is at most $2^{29} - 1$ bp (512 Mibp), so some files cannot be indexed.

Thus the standard indexing used with BAM and BCF is inapplicable and a custom method must be used.

BAM and BGZF splitting has been previously presented in Niemenmaa [Nie11] and the supplement to Niemenmaa et al. [Nie+12], and will thus not be covered in detail here. The basic idea is to use the magic numbers (present in BGZF only) and the inherent redundancies and consistency requirements in the data as something resembling impromptu synchronization markers. The equivalent strategy for BCF, implemented since Hadoop-BAM 6.0, is briefly presented below. Note that synchronizing to BGZF blocks is a separate issue from synchronizing to the underlying BAM or BCF data, and so the method used for BAM can be re-used for BGZF-compressed BCF.

Field name	Description	Type
`l_shared`	Length from start of CHROM to end of INFO	`uint32`
`l_indiv`	Total length of genotype fields	`uint32`
CHROM	Chromosome dictionary index	`int32`
POS	0-based co-ordinate	`int32`
`rlen`	Length of projected record (*ignored*)	`int32`
QUAL	Quality (*ignored*)	`float32`
`n_info`	INFO pair count	`int16`
`n_allele`	REF+ALT record count	`int16`
`n_sample`	Length of each `fmt_values`	`uint24`
`n_fmt`	Genotype field count (*ignored*)	`uint8`
ID	Identifier(s)	`str`
REF+ALT	Sequence strings (*ignored*)	`str[n_allele]`
FILTER	Indices into filter dictionary (*ignored*)	`vec`
INFO	Additional metadata (*ignored*)	`vec[n_info]`
	`n_fmt` genotype fields (all *ignored*)	
`fmt_key`	Identifier (*ignored*)	`int`
`fmt_type`	Type specifier (*ignored*)	`uint8`, optional `int`
`fmt_values`	`n_sample` values (*ignored*)	depends on `fmt_type`

Table 6.1: The format of the fields of one record in BCF. All integers are little-endian and floating point values are in the IEEE 754 [IEE08] format. `int` (when not followed by a bit width), `str`, and `vec` refer to the custom *typed* encodings used in BCF which are not detailed here. Fields that are not used by the algorithms presented here are marked as *ignored*. Note that the 'BCF2 site information encoding' table in the BCF specification [BCF] has QUAL in an incorrect position.

The binary layout of BCF records is shown in Table 6.1. The constraints and redundancies exploited in Hadoop-BAM are listed below. n_c and n_s refer to information found in the BCF header: the length of the chromosome dictionary and the number of samples, respectively.

1. The record length is sensible: $\texttt{l\_shared} + \texttt{l\_indiv} > 32$.
2. The chromosome dictionary index is valid: $\text{CHROM} \geq 0 \wedge \text{CHROM} < n_c$.
3. The positions and the two signed counts are nonnegative: $\texttt{POS} \geq 0 \wedge \texttt{n\_info} \geq 0 \wedge \texttt{n\_allele} \geq 0$.
4. The sample count matches to the value in the header: $\texttt{n\_sample} = n_s$.
5. The `ID` field should have a sensible type encoding and a reasonable length. Here i_0 and i_1 refer to the first two bytes of `ID`; l refers to the decoded length of the string, whose encoded size depends on i_0 and i_1; and & is a bitwise AND. The two constraints are as follows:
   a) $i_0 \,\&\, \texttt{0x0f} = \texttt{0x07}$
   b) If $i_0 \,\&\, \texttt{0xf0} = \texttt{0xf0}$, then:

$$(i_1 \,\&\, \texttt{0x0f}) \in [1, 3] \wedge l \geq 15 \wedge l \leq \texttt{l\_shared} - (32 + \texttt{n\_allele} + 2 \cdot \texttt{n\_info}).$$

Based on the above it is possible to find candidate locations: positions where a BCF record is likely to begin. A decoding test is then performed starting at each such location; once a certain number of records have been successfully read, it is assumed that the location was valid and should be used for the actual computation. An error at any point during decoding means that the next position should be tried.

Considerations for bioinformatical file format design

Existing bioinformatics-related file formats have not been designed with the requirements of distributed frameworks in mind, and so working with them tends to be needlessly difficult. The previously related experiences with BAM and BCF are representative of the kinds of issues that may arise. This Section presents a few considerations that ought to be taken into account when designing new row-oriented binary formats that may be used to hold Big Data sets.

Allowing for accurate splitting, without requiring something akin to the heuristic method that Hadoop-BAM employs for BAM and BCF, is of paramount importance. Synchronization markers are one possible method, with which a simple scan for a certain byte sequence can be used to find a record boundary. A more robust and performant method would be to provide, in the file header or footer, an exact index of the positions of certain records in the file. With an index, no scanning is required, as splits can be aligned ahead of time based on the contents of the index. In contrast to the standard BAM and BCF indexing scheme, indexed positions have to result in a sufficiently fine-grained partitioning for splitting to be effective: a simple solution would be to index the positions of every 100 000 records, or another similar number that is large enough to keep the index relatively small but small enough to also keep the minimum calculable split size suitably small.

Compression with speed-oriented schemes such as LZO or Snappy almost always improves I/O performance, which is often the bottleneck in distributed computing, at practically no cost and is thus worth implementing. However, directly applying a compressor to the file contents effectively neutralizes the effect of any synchronization markers or indices. Therefore compression should be implemented as part of the file format itself, keeping important metadata (such as what is needed for splitting) in the file uncompressed and compressing only the data records themselves. Additionally, because the output of most compressors is not splittable, the records should be divided into reasonably-sized blocks that are individually compressed.

Allowing users to choose the compression scheme, either per file or per block, can also be useful, as different compressors are optimal in different situations. When dealing with Big Data sets, even small percental differences between compression ratios can manifest as tens or hundreds of gigabytes of storage: selecting an appropriate compressor may be very important. Thus simply enforcing use of a decent but unexceptional compressor, such as the `gzip`-based BGZF in BAM and BCF, is not ideal.

6.2 RCFile

The RCFile (Record Columnar File) [He+11] format, is a hybrid storage format that has been implemented in Hive (and thereby Shark), Pig, and Impala (though Impala's support is currently, as of version 1.1.1, read-only), and was adopted as the default storage format by Facebook in its Hive-based data warehouse. RCFiles are a very simple realization of the hybrid storage

scheme, with only few special features in addition to the basic storage layout. These are summarized below.

- RCFiles contain a random 128-bit synchronization marker, just like SequenceFiles and Avro files, written at the start of each partition or *row group*.
- A run-length encoded [Cap59] metadata section, containing the number of records in the partition and the byte sizes of each column and each field in each column, is stored between the synchronization marker and the data columns of a row group.
- Compressed columns are decompressed on demand: when a row group is read into memory, each column within is kept in compressed form until its contents are needed for a query.

6.3 ORC

The ORC (Optimized Row Columnar) [ORC13; ORM] file format, or ORCFile, was designed to improve upon RCFile as a Hive-specific storage format. It was introduced as a part of Hortonworks's Stinger Initiative [StI]. Like RCFiles, ORC files use a hybrid storage layout, but with several unique features:

- ORC files do not contain synchronization markers. Instead, a footer at the end of the file (actually split into a compressed footer and a short, uncompressed *postscript*) stores the starting position of each partition or *stripe*. This way, instead of scanning for a marker, readers can jump directly to the correct position.
- The footer additionally stores a set of simple and commonly calculated statistics for each column: the number of non-null values, the minimum and maximum value, and the sum—as appropriate for the column type, e.g. strings are not summed, of course.
- Each stripe begins with an index that stores the positions of a subset of the records in the stripe. This facilitates seeking to a specific row in each column, which is useful e.g. when a filtering query selects only certain records. The index also contains the same statistics as the footer for the records contained in that stripe.
- The data for a column in a stripe may consist of multiple *streams* encoding different aspects of the values. Each stream is stored as

one consecutive, possibly compressed, sequence in the file. ORC files are aware of the types available in Hive and so can use an efficient type-specific encoding for each type. For example, a column containing strings may be dictionary encoded, using four streams containing:

1. For each unique string, the length of the string.
2. For each unique string, the actual contents of the string.
3. For each string, one bit identifying whether it is null.
4. An index into the stream of string contents, allowing retrieval of the data.

All streams are compressed using the generic algorithm named in the file footer, in addition to lightweight stream-specific compression such as run-length encoding. Unlike RCFile, which supports using any Hadoop-supported compressor, ORC currently supports three choices aside from not compressing at all: the DEFLATE algorithm—which it calls `zlib` presumably due to using the `zlib` library [Gai+13], not to be confused with the `zlib` format [Deu+96]—, Snappy, and LZO.

Using a footer, as opposed to an arguably more traditional header, for the stripe index is practically necessary due to the typically append-only file systems used and the large volumes of data that might be stored in a single ORC file. It would not be possible to reserve space for a header and fill it in after writing the data because that requires in-place modification, and because the amount of space reserved could be underestimated. Writing the header into a new file and then appending the data would, while possible, require copying all the written data and thus be too expensive. (A relatively cheap in-place append operation could be possible in a file system, but HDFS, at least, does not provide one, and so assuming its existence is not sensible.)

6.4 Trevni

Trevni [Tre13; TrG] is a hybrid file format that is currently part of the Apache Avro system and was created by Doug Cutting, who also originated Hadoop. Trevni is not yet integrated into any data warehouse system, but inclusion into Hive is planned [TrH12]. While mostly a simple realization of hybrid storage akin to RCFile, Trevni has some notable distinctions:

- Each row group is stored in a separate file. Each such file consists of only a single HDFS block—via setting the HDFS block size to a value

larger than the default, if necessary. Thus synchronization markers are not needed: each file should be allocated to only one task, which need not seek in order to get to its starting position.

- Columns in a row group are split into *blocks*. Block metadata at the start of each column indicates the row counts and the byte sizes of each block. This makes efficient seeking possible, like the position index in the stripes of ORC files.

- Optionally, the first value in each block can be stored in the block metadata. This can be used e.g. to speed up range queries in sorted data sets.

- A checksum of each block can be stored. When only few rows are queried, this can be cheaper than using the HDFS block checksumming. The only currently supported checksum algorithm is CRC-32, the 32-bit cyclic redundancy check [Pet+61], with the same field polynomial as used e.g. in the PNG (Portable Network Graphics) [Duc03] file format.

- Each column can be stored in a different compressed format. Two options are supported in the current version of Trevni: DEFLATE and Snappy.

- Trevni does not support the full set of types that Hive does, but it has special-case handling for arrays. One can define *array* columns, in which the values are prefixed by the array length. One can then define columns that refer to it as a *parent*, thereby sharing the length information without explicitly storing it.

6.5 Parquet

Parquet [Kes13; Par; Rya13] is yet another hybrid storage format, co-developed by Twitter and Cloudera as an improvement of Trevni. As of yet Parquet has been integrated only into Impala, with Hive support available as a separate download.

Parquet shares Trevni's one-to-one mapping of row groups to files and HDFS blocks, the splitting of columns into blocks (called *pages* in Parquet), the optional checksumming, and the ability to compress each column with a separate compressor—the options currently available are `gzip`, Snappy, and LZO. Parquet's additions include the following:

- Pages can have different encodings. (Compression is still specified at the column level.) There are currently two alternative encodings: a plain encoding, which packs Booleans so that they only use a single bit each and otherwise stores values as-is, and a dictionary encoding in which only indices to the dictionary are stored. The lookup table itself is stored as the first page of each column.
- An efficient columnar encoding of field data for records containing other, nested records, mixed with possibly repeated fields. For example, if a record contains an arbitrary number of integers, it is impossible to tell from a sequence of integers where the record boundaries are. While in this case storing a separate sequence of lengths for each record would be a suitable solution, in the presence of complicated nested structures with many levels of repetition such a simplistic encoding would not be very compact. The encoding used in Parquet is based on the technique applied in Dremel [Mel+10] and due to its complexity will not be detailed here. Note that the tabular data storage model of Hive implies that there is no nesting: this feature is only useful for other data models. Fortunately the overhead for flat data models like that of Hive tables is essentially zero.
- Various integers, such as those used to store indices in the dictionary encoding or those used in the nested field encoding, are either packed using the minimum required number of bits per integer, or run-length encoded. In each page, the more space-efficient encoding for the data in that page is used.
- Metadata, including among other things the positions of each column in the file, is stored in a footer, like in ORC files.

Chapter 7

Experimental procedure

> PLAN, v.t. To bother about the best method of accomplishing an accidental result.
>
> *The Devil's Dictionary*
> Ambrose Bierce, 1911 [Bie93]

The goal of the experiments performed was to investigate the performance of the Hive, Shark, and Impala frameworks in interactive sequencing data analysis tasks. Due to the plethora of file formats available, several different ones were also included as points of comparison, including alternative compression schemes. Other interesting features whose effect on performance could have been compared include indexing, table and column statistics, and partitioning. Unfortunately time constraints prevented their inclusion in these experiments.

The following Sections cover all of: the type of sequencing data used, how Hive support for it was implemented, and the data set itself; the originally intended benchmarks and their purposes; issues encountered during testing and benchmarking; the final benchmark set; and the setup, including the hardware and software environment used. The results obtained are presented and analysed in Chapter 8.

7.1 Accessing sequencing data

In order to access actual sequencing data, the appropriate file format support had to be implemented. Since Shark is Hive-compatible, a single implementation was used for both Hive and Shark. Impala's lack of extensibility means that implementing a new file format would require fa-

miliarity with its internals. In addition, the limited applicability of sequencing data formats means that they are not very useful to most Impala users. Thus such an implementation would likely have not been accepted into Impala proper, meaning that a low-level modification of Impala would have had to have been maintained indefinitely, or it would have eventually become useless due to its age. For these reasons, an implementation for Impala was not written; instead, Hive was used to convert the data into formats that Impala supports. The source code of the implementation described in the remainder of this Section can be found at `https://github.com/Deewiant/master-hive-bam/`.

The file format implemented was the BAM [SAM13] format. Due to time limitations, only this one format was implemented. There were a number of constraints which lead to the choice of BAM:

1. Sufficiently large data sets using the format had to be freely available.

2. The data in the format had to map to Hive tables reasonably well, and expressing interactive analysis tasks on the format in HiveQL had to be possible. User-defined functions should not have been necessary, so that Impala could also be tested.

3. The format had to be a compressed binary sequencing data format, These are more reasonably compared to the formats used in Hive, Shark, and Impala than the textual formats typically used, since they can be seen as reasonably 'optimized' given that they include their own compression and encoding schemes.

4. Ideally, the format had to be supported by Hadoop-BAM in order to minimize implementation work. Hadoop-BAM did not have BCF support when this work was begun, which, combined with the previous restrictions, made BAM the best choice.

Because of the additional complexities involved in outputting a correct BAM file, only input support was implemented. Regardless, as the benchmarked use case consists of exploratory analysis, outputting BAM is not important.

The BAM record layout is shown in Table 7.1. The brief descriptions of the fields are not intended to capture their full meanings; for a complete understanding, refer to the SAM specification [SAM13]. Note that the BAM header contains information that is needed to fully decode the record: for example, the `refID` field is an index to a dictionary contained in the header. In this work reading the header was not implemented, so such field data is treated as opaque.

Field name	Description	Type
block_size	Record length minus 4	int32
refID	Reference sequence ID	int32
pos	0-based co-ordinate	int32
l_read_name	Length of read_name	uint8
mapq	Mapping quality	uint8
bin	Bin number	uint16
n_cigar_op	Length of cigar	uint16
flag	Flags bit field	uint16
l_seq	Length of decoded seq	int32
next_refID	refID of next fragment	int32
next_pos	pos of next fragment	int32
tlen	Template length	int32
read_name	Name, null-terminated	uint8[l_read_name]
cigar	CIGAR string	uint32[n_cigar_op]
seq	Sequence data	uint8[(l_seq+1)/2]
qual	Sequence data quality	uint8[l_seq]
Auxiliary data until block_size is filled		
tag	Identifier	uint8[2]
val_type	Type specifier	uint8
value	Value	depends on val_type

Table 7.1: The format of the fields of one record in the BAM [SAM13] format. intN and uintN denote, respectively, two's complement signed and unsigned little endian integers with a bit width of N. x[N] denotes a length-N array of x.

The Hive table schema corresponding to BAM data, and applied in the implementation used in the experiments presented here, is given in Table 7.2. Instead of naïvely mapping the fields directly to their types' Hive equivalents, the schema is based on the SAM format. This choice was made for three main reasons:

1. The BAM data includes fields that are useful only for decoding it, such as `block_size` and `l_read_name`.

2. BAM files use reference sequence IDs, i.e. indices to the reference sequence dictionary found in the file header, instead of spelling out the name every time. While this is a significant space saving, it is an additional dependency on the header, for which a schema was not implemented, and columnar compression should mitigate the additional space usage incurred due to simply expanding the whole string every time.

3. Thirdly, many of the fields are of array type, and the corresponding ARRAY type of Hive is not supported by Impala [ImU]—to the extent that it refuses to run any queries on tables containing columns with unsupported types, whether or not those queries use such columns. Thus the string encodings of SAM were the only alternatives considered reasonable.

The auxiliary data fields in the schema are the only exception that do not adhere to the SAM format: they were grouped by their type, and each group was mapped to a single string. This was done because Impala does not support the MAP type. The intended schema would have had each group turned into a `MAP<SMALLINT,T>` with the appropriate value type `T` for each group. In order to approximate the data volume occupied by the auxiliary data, it was not dropped entirely, but a string encoding was used instead. Each group consists essentially of `tag=value` strings separated by the four-byte sequence $\{96, 0, 0, 96\}$.

Another notable difference between the original BAM data and the schema arises because Hive does not distinguish signed and unsigned integers: integers in Hive are always signed. Thus the `mapq` column must be used carefully in queries, lest seemingly negative values be mistreated. Using a larger integer type for the column would have avoided the issue, but would have instead allowed completely invalid values to be inserted into the column. Which of these is the 'lesser evil' is largely a subjective matter.

Column name	Column type
qname	STRING
flag	SMALLINT
rname	STRING
pos	INT
mapq	TINYINT
cigar	STRING
rnext	STRING
pnext	INT
tlen	INT
seq	STRING
qual	STRING
opts_char	STRING
opts_int	STRING
opts_float	STRING
opts_string	STRING
opts_arr_int8	STRING
opts_arr_int16	STRING
opts_arr_int32	STRING
opts_arr_float	STRING

Table 7.2: The Hive schema used for BAM data in the experiments presented here. The columns whose names begin with `opts` correspond to the various kinds of optional fields in SAM, and the preceding columns and their types correspond to the mandatory fields of SAM.

Mapping the fields into Hive in such a direct manner means that many useful operations require user-defined functions. For example, the sequence data itself, its quality metadata, and its comparison to the reference—the `seq`, `qual`, and `cigar` columns—are essentially opaque to normal Hive operations. Impala's limitations were once again a major factor: with this encoding, user-defined functions are necessary and so Impala is excluded, but the only other reasonable encoding, in which arrays instead of strings would be used for these columns, would also have excluded Impala, since it does not support the ARRAY type [ImU]. Because of Impala's inability to query tables containing columns with unsupported types, the nonprohibitive string encodings were chosen.

The data set used was a single BAM file from the 1000 Genomes Pro-

ject [1kG], with a size of about 301.2 GiB and exactly 986 759 806 records. Unfortunately, the file is no longer available on the 1000 Genomes web site. (Its URL was `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/NA12892/high_coverage_alignment/NA12892.mapped.ILLUMINA.bwa.CEU.high_coverage_pcr_free.20130520.bam`.) A newer replacement file of somewhat smaller size can be found at `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/NA12892/high_coverage_alignment/NA12892.mapped.ILLUMINA.bwa.CEU.high_coverage_pcr_free.20130906.bam`. In case of interest towards obtaining the original data set used, the e-mail address `matti.niemenmaa+master@iki.fi` may be contacted.

7.2 Intended procedure

The planned sequence of HiveQL queries mostly imitated actions that might be performed by an analyst interactively exploring a data set. Some statistics describing the complete data set were to be computed before narrowing it down to a significantly smaller 'previously chosen'—for these experiments, randomly generated—subset for further processing. The subset was given as a BED file [BED13], which could be directly loaded into a simple three-column table and combined with the full data set using a `JOIN` on the `rname` and `pos` columns. In addition, two more synthetic benchmarks were to be run: one to compare against the sorting capabilities of Hadoop-BAM by applying an `ORDER BY` query to the full data set, and another to test the performance of some aggregation functions such as `AVG` and `STDDEV_SAMP`.

While experimentation using bioinformatically meaningful queries would have been preferable, the limits of Impala combined with the quite simplistic SAM-based schema made this infeasible. In order to facilitate reasonable runtime comparisons, the HiveQL statements used with the different frameworks had to be as similar to each other as possible. Therefore introducing user-defined functions was not an option, making most meaningful operations practically impossible. Hence the queries consisted only of arbitrary, fairly simple actions.

The experiments were to be run with each of the following numbers of slave nodes: 1, 2, 4, 8, 16, and 31. Continuing the progression to 32 worker nodes was not possible, because one node was always allocated as the master and cluster policy prevented using more than 32 nodes at a time. In order to discover the amount of variance in the timings, the experiments were to be run four times for each combination of slave node count, framework,

and file formats.

7.3 Issues encountered

Throughout this and later Sections in this Chapter, the unqualified terms 'Hive', 'Shark', and 'Impala' refer to their specific versions used in these experiments: Hive 0.11, Shark 0.7, and Impala 1.0.1. (See Section 7.5 for more detailed version number information.) The issues discussed may have been corrected in later versions. It is explicitly noted if such later versions were released between the times when the experiments were run and this Thesis was completed.

None of Hive, Shark, and Impala implement a parallel `ORDER BY`: in the end, the data being sorted is always sent to a single node. Thus, the idea of comparing their sorting runtimes to Hadoop-BAM's was dropped. The frameworks would only have been impractically slow and likely caused out-of-memory errors and similar issues. In Hive, this issue [HIV10] has been addressed in version 0.12, but not in the 0.11 used in these experiments.

Impala does not support `STDDEV_SAMP` [ImU], nor any of the other aggregation functions that involve relatively advanced computations compared to e.g. `AVG` and `SUM`. The benchmark using `STDDEV_SAMP` was therefore changed to compute the standard deviation manually.

Shark was incapable of performing the join with the BED table without crashing due to running low on memory. Shark does contain an implementation of *map joins*, wherein the join would be done by propagating the smaller BED table to a number of tasks, each of which joins the entire BED table with a part of the larger BAM table. It was found that this implementation is not used by default, likely owing to Hive 0.9's inability to heuristically recognize map join opportunities by default. It was thought that Shark itself had appropriate heuristics for this—perhaps the implementation was not active in the used version, like partial DAG execution [SHA12], or the heuristics simply performed poorly in this case. In any case, the implementation of a standard join was completely impractical for the data set used: when computing joins that are not map joins, Shark assigns each join key value to one node, with all rows matching that key being sent to that node at once. Especially when coupled with the overhead of Java objects, this easily caused the memory budget to be exceeded.

To perform the join as a map join in Shark, the statement was manually annotated with a `MAPJOIN(bed)` hint [HiJ]. While now running without any fatal errors, Shark did not compute a correct result for the join, outputting zero rows instead of the expected 16 616 633. This was a previously

known issue [SHA13] which was unfortunately not corrected in time for the experiments run with Shark 0.7, but has since been fixed in Shark 0.8.

With only one worker node, Impala was also incapable of performing the BED join, instead aborting the query due to running low on memory. Fortunately two slave nodes were enough: only the single-worker runs were troublesome.

After converting the data set to the Parquet format in Impala, accessing the result with Hive was attempted. This resulted only in error messages, and so testing Parquet under Hive was not included in the experiment. The later released Impala 1.1.1 includes a fix for this issue [ImF]. Older versions of Impala, including the used 1.0.1 version, output Parquet files containing metadata that is incompatible with Hive. Unfortunately this was neither realized nor corrected in time for the experiments.

An Impala issue concerning Snappy-compressed RCFiles was discovered and reported [ImB13]. While quite quickly marked as fixed, the code implementing the fix was not made public for another month [IMP13] and so was not available for the experiments. (Embarrassingly, the fact that not using null bytes in the optional field encoding would have worked around the problem was not realized in time.) As with the Parquet-related issue, the fix is part of the Impala 1.1.1 release, but not the used 1.0.1.

Finally, Impala seemed to suffer from some kind of stability issue in RCFile processing when run in configurations with 4, 8, or 16 worker nodes. Crashes manifested seemingly randomly, with runs sometimes succeeding and sometimes failing. In the end, the failure rate with 16 slave nodes was too high to obtain the desired four successful runs with each configuration, and so Impala's 16-worker runs were replaced with runs using only 15 slaves. Unfortunately, by the time the severity of this issue became evident, all of the other frameworks' 16-worker runs had been completed, so due to time constraints only the 16-worker runs of Impala were replaced. While no bug report for this issue has been filed, brief testing with Impala 1.1.1 suggests that the issue is no longer present.

In summary, the issues encountered prevented all of the following:

- Benchmarking the ORDER BY capabilities of each framework compared to the BAM sort of Hadoop-BAM.
- With Shark, and Impala on a single-worker 'cluster', properly benchmarking the join with the BED table and the subsequent operations.
- Using the Parquet file format in Hive.

- With Impala and the Snappy-compressed RCFile format, benchmarking the join with the BED table and the following operations.
- Using clusters of 16 worker nodes with Impala: 15 workers had to be used instead.

7.4 Final procedure

Three groups of HiveQL statements ended up comprising the benchmarks. These groups were as follows:

1. Table creation and possible settings needed for the file format, such as compressor selection. One version of this group was created for each combination of file format and compression scheme.
2. Queries run on the full data set, including the `JOIN` statement with a BED file randomly generated for this purpose.
3. Exploratory queries on the resulting smaller data set.

The table creation statements were run separately so that they could be easily changed without modifying the queries, thus forming the different benchmarks. Another reason was that then the table data could be copied into place directly, speeding up the data loading process. The queries on the reduced data set were separated due to the issues with Shark discussed in Section 7.3. (Originally the `JOIN` was not run on Shark at all due to producing an incorrect result, but later it was enabled, as it was realized that its runtime may still indicate something useful.) All HiveQL statements used are listed in Appendix B; Listings 7.1 to 7.3 below show the statements used for Hive with the RCFile format and `gzip` compression.

The statements shown in Listing 7.1 mainly create tables following the schema given in Table 7.2. Two similar tables are created: one for the full data set, called `bam`, and one for the reduced data set computed by the join, called `results`. A table for the BED file, called `bed`, is also created. Note how BED data can be read directly, as it is a simple row-oriented textual format with fields separated by horizontal tab characters. Finally, appropriate settings for `gzip`-compressed RCFile output, including a split size recommendation from Cloudera [IRC], are applied.

After creating the tables, their data was copied directly into place by a separate process—this being possible because converting the original BAM data into the various file formats was done in advance. Loading the data separately instead of with `LOAD` statements allowed using the

```
CREATE TABLE bam (
   qname STRING, flag SMALLINT, rname STRING, pos INT,
   mapq TINYINT, cigar STRING, rnext STRING, pnext INT,
   tlen INT, seq STRING, qual STRING, opts_char STRING,
   opts_int STRING, opts_float STRING,
   opts_string STRING, opts_arr_int8 STRING,
   opts_arr_int16 STRING, opts_arr_int32 STRING,
   opts_arr_float STRING
) STORED AS RCFILE;
CREATE TABLE results
   -- The rest omitted: it is identical to the above.
CREATE TABLE bed (
   chrom STRING, chromStart INT, chromEnd INT)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
   STORED AS TEXTFILE;
SET hive.exec.compress.output=true;
SET mapred.max.split.size=256000000;
SET mapred.output.compression.type=BLOCK;
SET mapred.output.compression.codec=
   org.apache.hadoop.io.compress.GzipCodec;
```

Listing 7.1: HiveQL statements used in Hive to create the tables in the RCFile format and to prepare for compressing output with `gzip`.

Hadoop DistCp (distributed copy) [DCp] tool to speed up the copy into HDFS. DistCp was used for each file format except BAM, because they split the data set into hundreds of files, making the file-level parallelization of DistCp effective. Care was taken to preserve the larger-than-default HDFS block sizes in the Parquet and ORC files: each file was stored in a single HDFS block.

The HiveQL statements concerning the full data set, shown in Listing 7.2, begin with a further setting: the number of reduce tasks R, which was set to four times the number of slave nodes available. The sequence of queries starts with fetching the counts of variously flagged records. The following queries compute two sorted histograms of the data: first, the number of records referring to each reference sequence, and second, for each possible mapping quality value, the number of non-duplicate records with valid qualities having that value. The `PMOD` function, which computes the positive remainder of its first argument divided by its second argument,

```
SET mapred.reduce.tasks = R;

SELECT COUNT(*) AS total FROM bam;
SELECT COUNT(*) AS mapped FROM bam WHERE flag & 4 = 0;
SELECT COUNT(*) AS passedQC FROM bam
   WHERE flag & 512 = 0;
SELECT COUNT(*) AS notDuplicate FROM bam
   WHERE flag & 1024 = 0;

SELECT rname, COUNT(*) FROM bam
   GROUP BY rname ORDER BY rname;

SELECT PMOD(mapq,256) AS pmapq, COUNT(*) FROM bam
   WHERE flag & (4 | 1024) = 0
   GROUP BY mapq ORDER BY pmapq;

INSERT OVERWRITE TABLE results
   SELECT DISTINCT
      -- All columns in bam, omitted for brevity.
   FROM bed JOIN (SELECT * FROM bam WHERE
                     flag & 4 = 0 AND seq <> "*") bam
              ON bam.rname = bed.chrom
   WHERE bam.pos                    <= bed.chromEnd
     AND bam.pos + length(bam.seq) >= bed.chromStart;
SELECT COUNT(*) FROM results;
```

Listing 7.2: HiveQL statements used on the full data set—the `bam` table—in Hive. R was replaced by four times the number of slave nodes available.

is used so that the histogram shows the numbers in the range $[0, 255]$ instead of $[-128, -127]$, thereby also ordering them correctly. These queries were intended as simulating a user viewing some of the essential characteristics of the full data set.

The penultimate query in Listing 7.2 is the join with the BED data. It involves several non-obvious aspects:

- The `flag` condition filters out BAM records with invalid co-ordinate data.
- A filtering condition on `seq` is required for correctness because the

length of the `seq` string is used to find the ending co-ordinate of the range covered by the record. A more typical way would have been to compute the length from the CIGAR string, since it is more commonly available, but without user-defined functions this would have been extremely complicated.

- Because Hive (as well as Shark and Impala) only support equality expressions in the join condition, the join itself is performed only on the reference sequence name, with the more important co-ordinate overlap checks performed as ordinary `WHERE` conditions.
- Since each BAM record may match to more than one co-ordinate range in the BED file, the `DISTINCT` feature is used to remove duplicates from the final result.

A count is performed after the join, and after each following `INSERT` operation shown in Listing 7.3. This was intended to mimic a user performing different operations in an attempt to diminish the data set size until it could be more closely examined in other tools.

After reducing the data set to that covered by the BED table, the HiveQL statements in Listing 7.3 were run. First another join is computed, this time to select only the highest-quality reads covering each co-ordinate range. The read with the highest mapping quality for each `pos` and `tlen` is selected. Once again a `PMOD` is necessary so that `MAX` compares the values in the right way and finds the correct maximum. Next, three simple filters are applied, performing the final reductions on the data set. Note that the last filter completely subsumes the second-to-last one: such sequences of operations can legitimately arise when working interactively, though they would be deemed mistakes in a non-interactive context.

Finally, the mean and sample standard deviation of the `tlen` column values are computed. This is done in one statement using a join with the mean, to work around Impala's lack of support for `STDDEV_SAMP`.

The changes in the size of the data set through the join with the BED data and the reductions in Listing 7.3 are shown in Table 7.3. Clearly, the BED join performs the greatest reduction on the data set size. The following updates are insignificant in comparison. In particular, the last two selections do not change the data set size at all—thus, not only is the next-to-last query irrelevant in the face of the last, but in fact they both have no effect. This, too, is a likely possibility when working interactively. In theory, a sufficiently smart implementation may compute a histogram of the `LENGTH(seq)` values while running the first of these two queries, thereby recognizing the filtering expression in the second as vacuously true

```
INSERT OVERWRITE TABLE results
   SELECT results.*
   FROM (SELECT pos, tlen, MAX(PMOD(mapq,256)) AS maxq
            FROM results
            WHERE flag & 4 = 0 AND mapq <> -1
            GROUP BY pos, tlen) tmp
      JOIN results ON results.pos  = tmp.pos
                  AND results.tlen = tmp.tlen
   WHERE PMOD(results.mapq,256) = tmp.maxq;
SELECT COUNT(*) FROM results;

INSERT OVERWRITE TABLE results SELECT * FROM results
   WHERE mapq <> -1 AND PMOD(mapq,256) >= 60;
SELECT COUNT(*) FROM results;
INSERT OVERWRITE TABLE results SELECT * FROM results
   WHERE LENGTH(seq) >= 10;
SELECT COUNT(*) FROM results;
INSERT OVERWRITE TABLE results SELECT * FROM results
   WHERE LENGTH(seq) >= 50;
SELECT COUNT(*) FROM results;

SELECT AVG(tlen),
       SQRT(SUM(POW(tlen - mean, 2)) / (COUNT(*) - 1))
   FROM (SELECT AVG(tlen) AS mean FROM results) tmp
        JOIN results;
```

Listing 7.3: Exploratory HiveQL queries used on the reduced data set—the `results` table—in Hive.

immediately. While such a result was not expected, the queries were left as they are in order to draw attention to such possibilities.

In Shark, since the join with the BED data was not computed correctly, the statements of Listing 7.3 could not sensibly be run immediately following the join. Instead, the result of the join as computed by Hive was saved, and then loaded separately for the Shark benchmarks. This of course may have perturbed the resulting runtimes due to effects on e.g. RDD persistence in Spark or file caching at the operating system level, but it was the only way of performing the correct computations in Shark. The same procedure was also used for Impala's single-worker runs.

Modification	Data set size (rows)	Data set size (percentage of previous)
Initial	986 759 806	100.00%
Joined with BED	16 616 633	1.68%
Selected best `mapq`	16 546 449	99.58%
Selected `mapq` $\geq$ 60	15 153 080	91.58%
Selected `LENGTH(seq)` $\geq$ 10	15 153 080	100.00%
Selected `LENGTH(seq)` $\geq$ 50	15 153 080	100.00%

Table 7.3: The size of the data set initially, and after each time it is modified in HiveQL.

Many of the statements had to be tweaked for the different frameworks due to differences in HiveQL support. The various changes that needed to be made to the code in Listing 7.2 consisted of the following:

- Impala has no equivalent to the `mapred.reduce.tasks` setting: the degree of parallelism is always selected automatically, so no such setting was applied.
- Impala requires a `LIMIT` in any query involving `ORDER BY`. For the `rname` histogram, this limit was set to 100 after checking manually what an appropriate limit would be. For the `mapq` histogram, the limit was set to 256: the maximum possible number of different `mapq` values.
- As previously discussed: in Shark, the `MAPJOIN(bed)` hint was applied to the `JOIN` statement, and before realizing that timing even the incorrect result may be useful, the statement was completely disabled. Thus only partial results for this statement are available in Chapter 8.
- While Hive's optimal join ordering requires that the tables be joined in ascending order [HiJ], in Impala this order is reversed [ImL], and so the `bed` table was mentioned last, not first, in the `JOIN` statement.

The changes required for the code in Listing 7.3 were as follows:

- As above, the order of the tables in the two `JOIN` statements was reversed in Impala.

- Impala refuses to execute a `JOIN` statement without an equality predicate [ImL], such as the one used here to compute the sample standard deviation. The argument that such joins could result in excess resource usage does not apply here, as the 'table' joined with consists of only one column and one row: the `AVG(tlen)` value. To work around this limitation, a dummy column whose value was always zero was added to each side of the join, and those columns were used in an equality condition. Note that Impala saw through simply adding the constant `0`, so simple subtractions resulting in zero were used instead: `tlen-tlen` and `COUNT(tlen)-COUNT(tlen)`.

The details of the changes are evidenced in Appendix B.

7.5 Setup

Table 7.4 lists all combinations of file formats, compressors, and frameworks benchmarked. Because the frameworks cannot always output to the same format as the format of the input data set, the format for the `results` table is listed separately. For example, Impala can only write in textual or the Parquet format, and thus, in these experiments, the `results` table in any Impala-using benchmark run is always stored in the Parquet format, regardless of the format of the `bam` table.

In an effort to see how the BED join query would behave, different compressors were used for both tables with RCFile. For Parquet, only the Snappy compressor was used, because it was the only one supported by Impala. Similarly, because the version of Shark used is based on Hive 0.9, it does not provide built-in support for Snappy, and so only `gzip` was used. Hive had no significant limitations: when testing it with RCFile and ORC, both Snappy and the supported DEFLATE-based compressor were used, respectively as representatives of relatively fast and relatively slow compressors.

The used versions of all relevant software involved were as follows:

- Hadoop 2.0.0 from Cloudera's distribution, CDH 4.3.0. Additionally, the native library [HNL13] was compiled in the hopes of achieving greater performance. YARN was not used.
- Hive 0.11.0.
- Spark 0.7.3 and Shark 0.7.0, running on Scala 2.9.3. Shark additionally used its bundled Hive 0.9.0.

`bam` table		Framework	`results` table	
Format	Compressor		Format	Compressor
BAM	BGZF	Hive	RCFile	`gzip`
				Snappy
			ORC	DEFLATE
				Snappy
		Shark	RCFile	`gzip`
RCFile	`gzip`	Hive	RCFile	`gzip`
				Snappy
		Shark	RCFile	`gzip`
		Impala	Parquet	Snappy
	Snappy	Hive	RCFile	`gzip`
				Snappy
		Shark	RCFile	`gzip`
		Impala	Parquet	Snappy
ORC	DEFLATE	Hive	ORC	DEFLATE
	Snappy	Hive	ORC	Snappy
Parquet	Snappy	Impala	Parquet	Snappy

Table 7.4: The experiment plan: the file formats, compressors, and frameworks used, with separate formats and compressors for the `bam` and `results` tables.

- Impala 1.0.1.
- Hadoop-BAM 6.0.
- PostgreSQL 9.2.4, which was used as the metastore.
- Snappy 1.1.0 and `zlib` 1.2.3, which implements both `gzip` and plain DEFLATE. Snappy was compiled manually, while `zlib` was provided by the Linux distribution used—Scientific Linux 6.4.
- Dstat 0.7.0 [Wie], which was run together with the experiments to observe resource utilization in detail.
- Java Standard Edition Development Kit 7u25, and the runtime environment included with it. Since the frameworks were available as pre-built binaries, the development tools were used only to compile

the BAM input support for Hive and Shark, and when building the native Hadoop libraries. The runtime environment was used to run all software based on Java or Scala.

- The GNU Compiler Collection, version '4.4.7 (Red Hat 4.4.7-3)', which was used to build the native Hadoop library as well as the Snappy libraries.
- Python 2.6.6, which ran at least Dstat as well as parts of the Impala frontend.
- Linux 2.6.32-358.14.1.el6.x86_64 and `glibc` 2.12 were the kernel and base C library installed on the nodes.

The hardware used was a larger cluster's 100-node subset with nodes composed of the following main components:

- Two six-core Intel Xeon X5650 processors clocked at 2.67 GHz.
- 48 GiB of DDR3 SDRAM [DDR12] main memory clocked at 1066 MHz.
- 4x QDR Infiniband network connections, for an aggregate 40 Gbps of theoretical maximum throughput.
- About 837 GiB of usable local disk space, striped across two 7200 RPM hard disk drives. Be aware that because of uneven disk performance this striped configuration may be suboptimal compared to treating the disks as separate volumes, in what is commonly called a JBOD (just a bunch of disks) configuration [Kre+11].

In addition, log files were written to a version 1.8.7 Lustre [Lus; Sch03] file system served by a DataDirect Networks SFA10K system [DDN] with four storage servers and two metadata servers.

The experiments were mostly run as described in Section 7.2: with 1, 2, 4, 8, 16, and 31 worker nodes, and four times for each combination of node count, framework, and storage formats. The main exception to this came about due to the previously mentioned Impala issue forcing runs with 15 worker nodes instead of 16. Other than that, one 31-slave run in each of the configurations shown in Table 7.5 was not run (i.e. did not receive a resource allocation in the queueing system used on the cluster) in time for this publication and thus their results are unavailable. As each combination was to be run four times, these losses are not problematic: they only mean that for some combinations, only three sets of results were obtained.

`bam` table		Framework	`results` table	
Format	Compressor		Format	Compressor
RCFile	Snappy	Hive	RCFile	`gzip`
				Snappy
		Shark	RCFile	`gzip`
		Impala	Parquet	Snappy
ORC	DEFLATE	Hive	ORC	DEFLATE
Parquet	Snappy	Impala	Parquet	Snappy

Table 7.5: The experimental configurations for which a set of 31-node results could not be completed due to time constraints. In each case, precisely one result for every applicable query is missing.

Various important configuration settings are listed below. The almost complete settings used, excluding only environment-specific information such as precise file paths and network host names, are available in Appendix A.

- The HDFS replication level was set to $\max(3, \lfloor n/2 \rfloor)$, where n was the number of slave nodes used.
- The number of map and reduce slots per node were set to 12 and 6, respectively.
- Map output compression using Snappy was enabled, meaning that the output of map tasks was compressed with Snappy before being sent to the reduce tasks.
- In accordance with the configuration settings recommended in the Impala documentation [ImC], short-circuit reads and block location tracking were both enabled.

Chapter 8

Experimental results

> Science invites us to let the facts in, even when they don't conform to our preconceptions.
>
> 'Why We Need To Understand Science'
> Carl Sagan [Sag90]

This Chapter is divided into two Sections. The first Section discusses the size of the data set when stored with different formats and compressors. The second Section examines query performance across both the frameworks and the storage formats.

8.1 Data set size

Table 8.1 shows the size of the data set in each of the format-compressor combinations benchmarked and in variously compressed SAM formats. The sizes are compared to BAM, as it is the *de facto* binary storage format for this kind of sequence data and the original format used by the source of the data, the 1000 Genomes Project [1kG]. The remainder of this Section is focused on analysing Table 8.1.

Two things concerning the information in Table 8.1 should be noted. Firstly, while the header data is only included in the SAM and BAM formats due to the schema used, its presence has barely any effect in this comparison, because it consists of only 3515 bytes when uncompressed—less than a millionth of the data set size in any of the formats. Secondly, all the `gzip` compressors, including BGZF, used the default level of compression, striking a balance between data size and compression speed. Since the decompression speed of `gzip` is likely to only improve as a higher level of

Format	Compressor	Size (bytes)	Size (relative)
SAM	none	1 363 445 233 534	421.60%
	`gzip`	291 484 653 668	90.13%
	BGZF	314 945 209 503	97.39%
BAM	none	1 217 084 188 364	376.35%
	`gzip`	295 794 591 321	91.47%
	BGZF	323 395 742 321	100.00%
RCFile	`gzip`	243 640 049 627	75.34%
	Snappy	466 314 416 496	144.19%
ORC	DEFLATE	344 239 681 323	106.45%
	Snappy	586 681 615 929	181.41%
Parquet	Snappy	485 801 576 705	150.22%

Table 8.1: The size of the data set in different file formats and with different compressors.

compression is used [Col05], this was nothing but a pessimization. Except for the BGZF-compressed BAM file, which was provided by 1000 Genomes, this came about as the result of assuming that Hive, which was used to create the `gzip`-compressed RCFile data set, would default to the highest level of compression, an assumption that was found to be false only when it was too late to re-run the experiments. For the sake of fair comparison, the same compression level was then applied to SAM and BAM resulting in the values in Table 8.1.

Among the first results visible in Table 8.1 is the fact that the data set was slightly smaller in SAM than in BAM when the two were compressed with the same compressor, even though BAM was smaller than SAM when uncompressed. This raises the question of whether performing computations on compressed SAM would also be slightly faster, which would make SAM superior to BAM for all practical purposes. Unfortunately this investigation was not performed in these experiments.

Another result shown is that BGZF was about 8% less effective than the `gzip` it is based on. This can be explained via the limitation imposed by BGZF that compressed blocks be at most 64 KiB in size. While the difference in compression ratios is not a new result in itself, its magnitude has previously been assumed to be smaller than this for sequencing data [Coc11].

When it comes to the tabular formats, an unexpected result seen in Table 8.1 is that RCFile was smaller than both ORC and Parquet. While the

difference between the RCFile and Parquet sizes was not particularly large, and can be explained with Parquet's additional complexity and metadata, using ORC results in the largest data set sizes by far. This contradicts previous results comparing ORC to RCFile [OMa13] in which the data set of the TPC-DS [TPC] benchmark was smaller in ORC than in the RCFile format. One possible, though highly questionable, explanation is that Cloudera's recommended settings for compressed RCFile output [IRC] simply caused that large a difference, assuming that the previous results used much less optimized settings. A more likely possibility is that ORC somehow copes poorly with the kind of sequencing data found in BAM files, perhaps in particular when combined with the schema used in the tabular formats. ORC is still a new technology, so the presence of such a problem is plausible. Future improvements are possible and may mitigate the issue.

A final note regarding the size data is that compressors optimized for speed can indeed result in noticeably larger sizes than even fairly balanced compressors: Snappy compression gives approximately 170% of the `gzip`-compressed size with RCFile and about 191% of the DEFLATE-compressed with ORC. Thus, as the differences can be highly significant, selecting the appropriate compressor based on one's needs is very important.

8.2 Query performance

The full query performance data presented and analysed in the remainder of this Section can be obtained from `https://github.com/Deewiant/master-data/`. In addition to the timings themselves, the output of Dstat as well as Impala's per-query profiles are included.

Overviews by framework

Figures 8.1 and 8.2 show overviews of query performance, displaying the same data points but with linear and logarithmic time scales respectively. Each point corresponds to the execution of a single query in a certain framework. To enable comparing the different frameworks, their data points are slightly separated from one another at each X-axis position. Note that the timings of Shark's BED joins, in spite of the incorrect results produced, are included in these Figures (and later Figures where appropriate).

It is clearly visible in Figure 8.2 that Hive was incapable of executing any query in less than ten seconds. This matches with the previously known

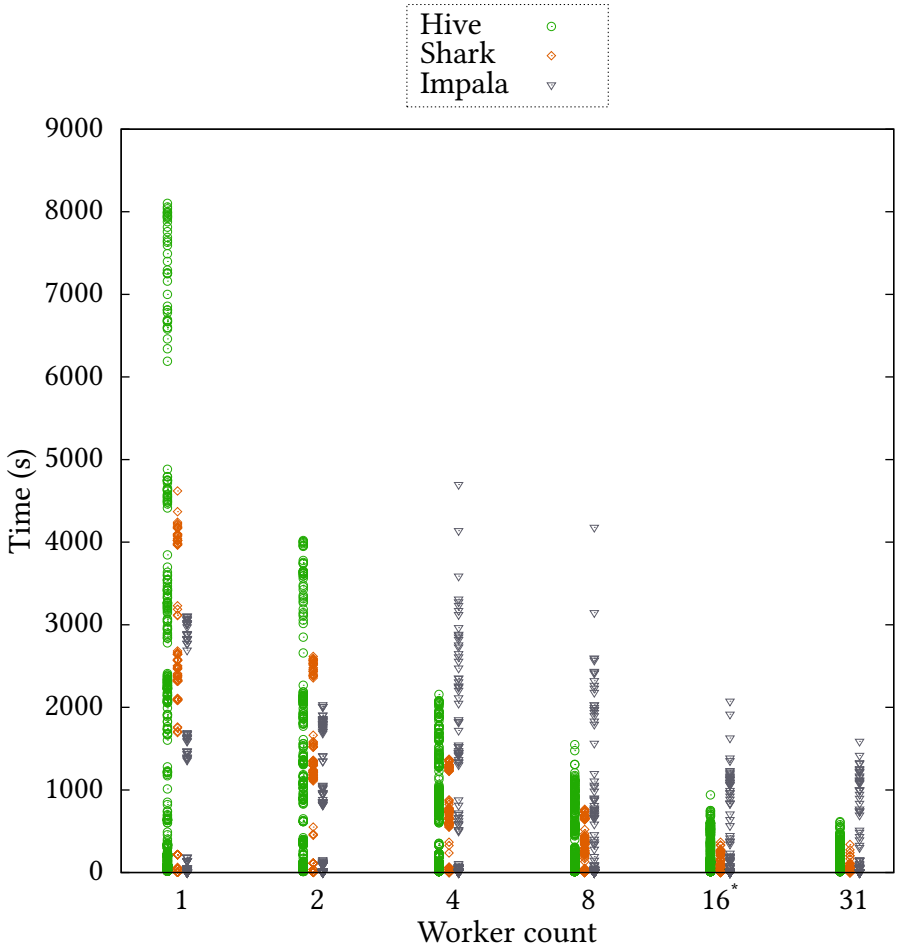

Figure 8.1: The runtime of each individual query shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the execution framework.
*For Impala, 15 workers were used instead of 16, but those runs are considered replacements for the 16-worker runs and are thus grouped as such.

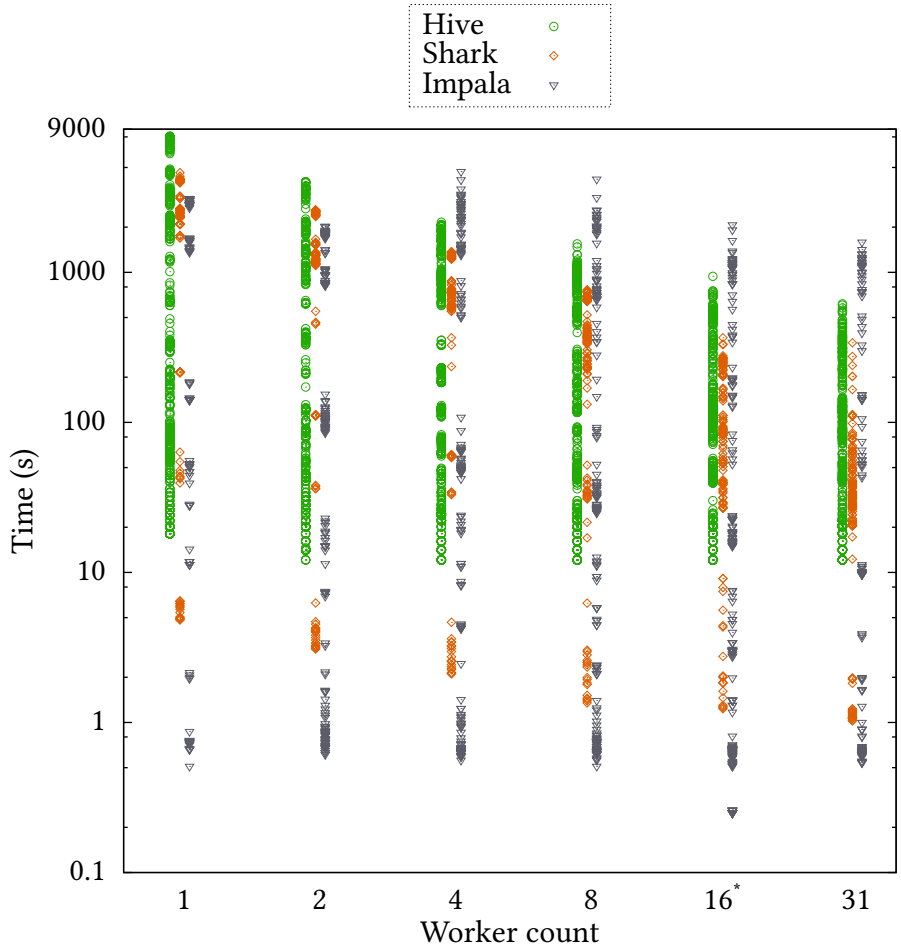

Figure 8.2: The runtime of each individual query shown on a logarithmic scale compared to the amount of worker nodes available, with the data points separated by the execution framework.
*For Impala, 15 workers were used instead of 16, but those runs are considered replacements for the 16-worker runs and are thus grouped as such.

result that the startup time of a MapReduce job is about ten seconds [Pav+09; Xin+12]. This lower bound is arguably Hive's main weakness in interactive use. Satisfyingly, Shark managed to complete many queries in as little as one second, and Impala often reached the half-second mark.

For the most part, all three frameworks achieved decent speedups with increasing worker node counts on the longer-running queries. The more quickly finishing queries reached the 10 s lower limit in Hive and a seeming 0.5 s lower limit in Impala already at the two worker node mark, and so did not speed up similarly well overall. With Impala's extremely fast speed, however, this can hardly be considered an issue. Shark seemed to achieve speedups in the faster queries all the way up to the maximum of 31 worker nodes used, although the returns diminished rapidly after eight worker nodes.

The stability issue affecting Impala in RCFile processing, described in Section 7.3, clearly also had an effect on the runtimes even when crashes did not occur. Figure 8.3 shows the runtimes of only the queries run on the `bam` table by Impala, with the different storage formats used for the table separated for comparison. Therein all the queries seem to speed up quite naturally between one and two worker node clusters. However, the RCFile runtimes were chaotic when four or eight worker nodes were used, with several queries being much slower than even with just one worker node. The crashing issue only affected those two slave node counts. But there also appear to have been other issues with Impala's RCFile implementation, given the poor and often negative speedup observed between 2 and 15 or even 31 workers.

As is most evident in Figure 8.1, Hive tended to have the longest-running queries until the 16-slave mark, after which point Impala definitely failed to keep up. This analysis is muddled by Impala's RCFile processing issue causing great slowdowns after the two worker node mark; were it not for that issue, it could likely be conclusively stated that Hive was the slowest of the frameworks.

Overviews by storage format

Hive has the widest level of format support of the frameworks benchmarked, enabling the rich set of storage format comparisons presented in Figures 8.4 to 8.7. Figures 8.4 and 8.5 exhibit the queries run by Hive on the `bam` table with linear and logarithmic scales respectively, while Figures 8.6 and 8.7 latter portray those run on the `results` table, also with linear and logarithmic scales respectively. Note that the results of running the BED

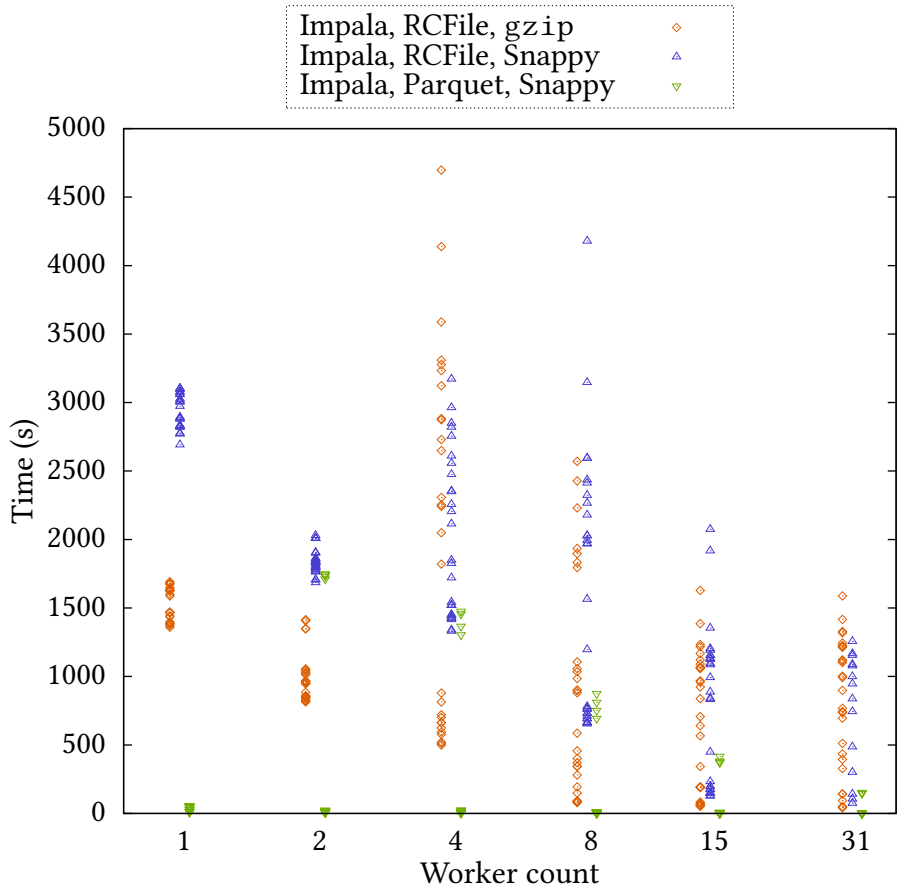

Figure 8.3: The runtime of each query run by Impala on the `bam` table. The data is shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `bam` table.

`JOIN` statement, which is affected by the format of both tables, are included in all four Figures.

From Figures 8.4 and 8.5 it can be seen that BAM did not perform particularly poorly. While not allowing for the fastest runtimes, it was also not the slowest of the formats. Among these results, the runtime differences between the BAM and `gzip`-compressed RCFile formats appear to largely follow the differences in the corresponding data sets' sizes, with `gzip`-compressed RCFile tending to have been about 25% faster, just as it was smaller—recall Table 8.1. Apart from the BED join (mostly clearly visible in each Figure as a separate set of slower points at any worker count), Snappy-compressed RCFile was similarly about 40% slower than

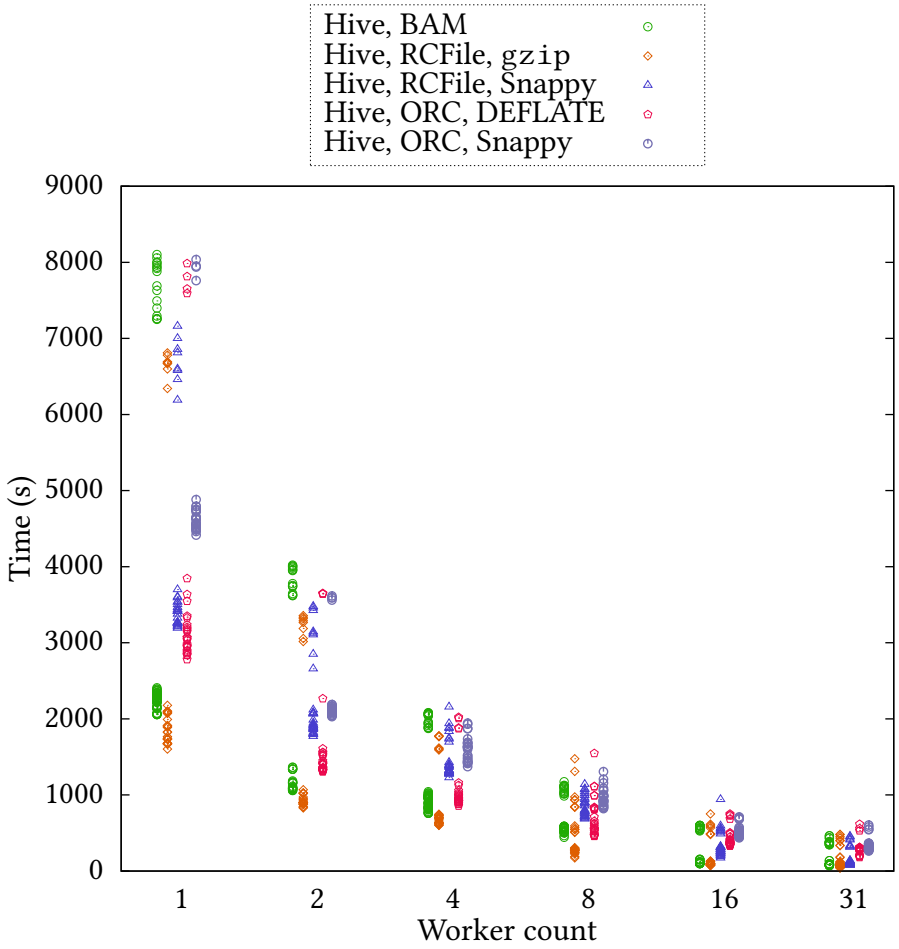

Figure 8.4: The runtime of each query run by Hive on the `bam` table. The data is shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `bam` table.

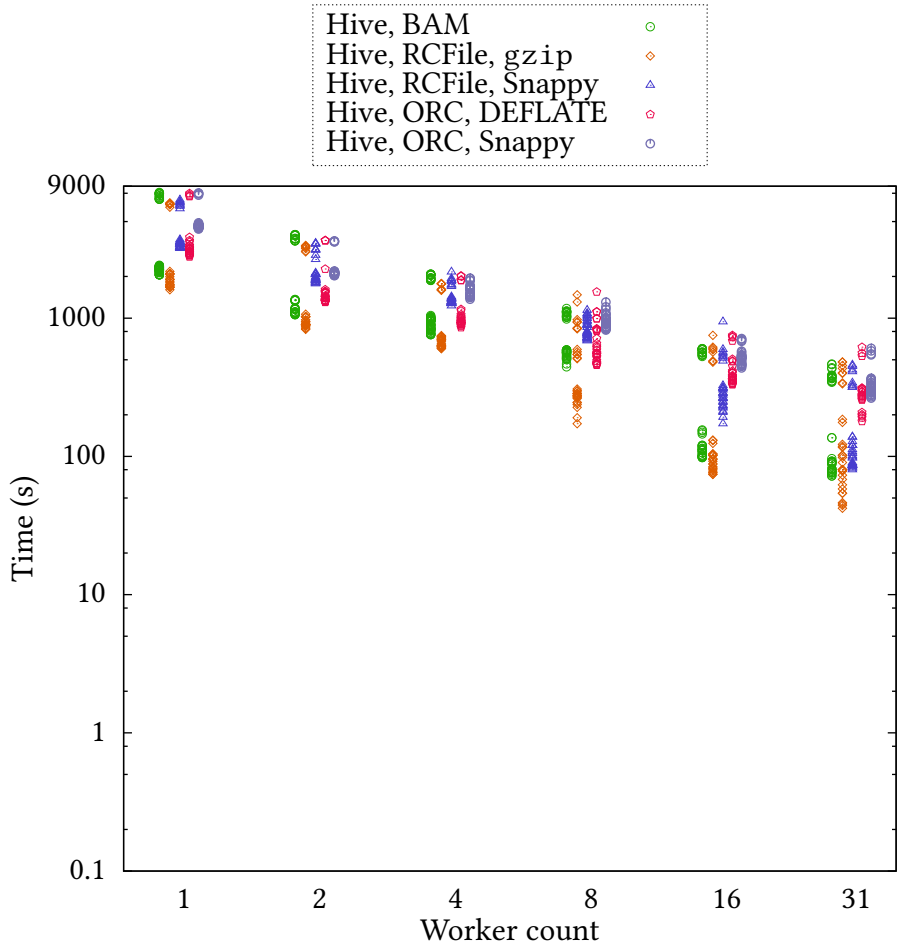

Figure 8.5: The runtime of each query run by Hive on the `bam` table. The data is shown on a logarithmic scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `bam` table.

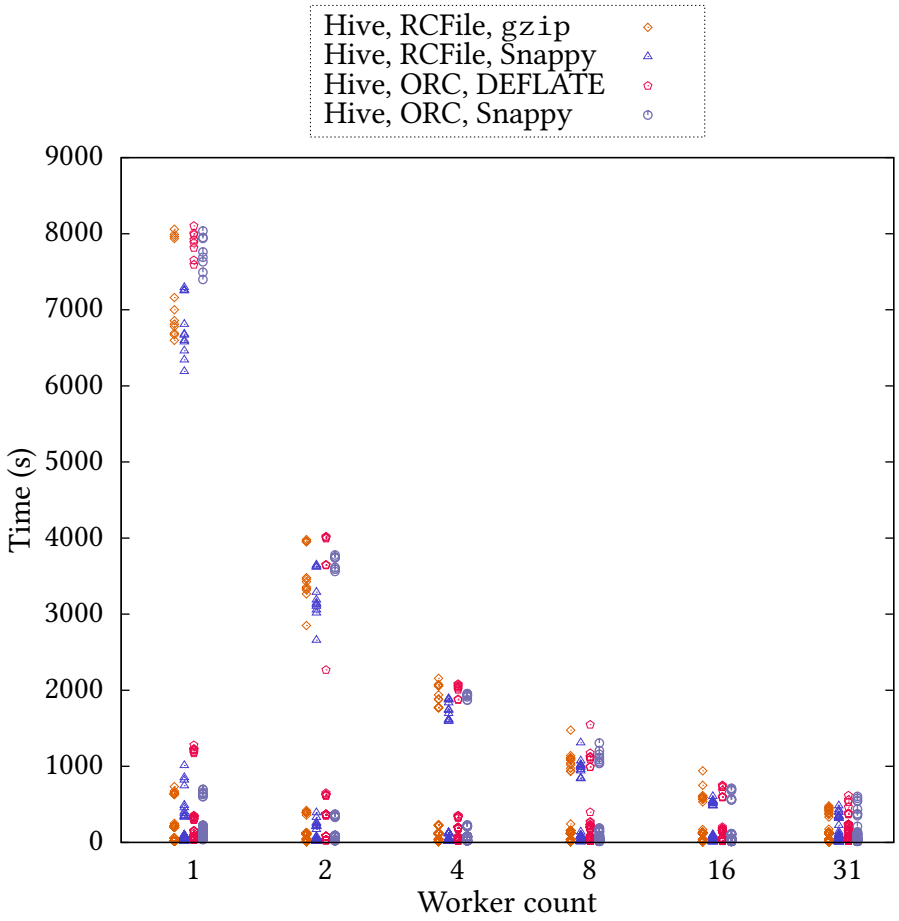

Figure 8.6: The runtime of each query run by Hive on the `results` table. The data is shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `results` table.

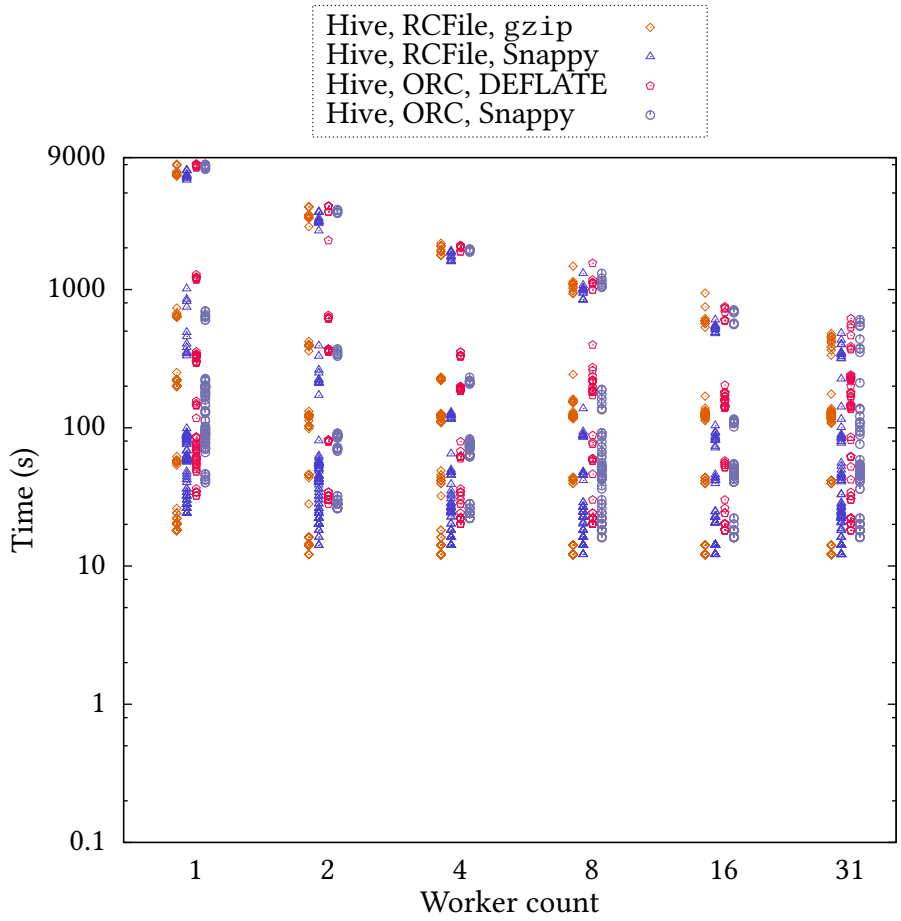

Figure 8.7: The runtime of each query run by Hive on the `results` table. The data is shown on a logarithmic scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `results` table.

BAM. However, the BAM runs seemed to suffer from diminishing returns after 16 worker nodes, bringing the Snappy-compressed RCFile format approximately even with BAM in the end.

These results are likely to be partially due to the very CPU-heavy worker nodes used. Compared to Cloudera's recommendations for Hadoop clusters [ODe13], the worker nodes used in this experiment were laughably disk-light, having only two hard disk drives for twelve CPU cores. Even for a 'Compute Intensive Configuration' in which disks are of lesser importance, Cloudera suggests four to eight disk drives for twelve CPU cores. This explains why the DEFLATE-compressed formats were very competitive with, and often better than, Snappy-compressed formats: the CPUs were waiting on I/O so much that the more computationally intensive compression could be performed with no trouble.

Also visible in Figures 8.4 and 8.5 is that ORC continued to perform poorly. Not only were its data sizes unexpectedly large, but its query runtime performance was also sub-par. ORC was never the fastest format here, although it sometimes was the slowest. Its greater data set size partially explains this phenomenon, but not completely: at the 16 and 31 worker node marks, BAM produces noticeably faster runtimes than ORC, despite only an approximately 6% difference in their data set sizes.

In Figures 8.6 and 8.7 it can be seen that with this far smaller data set in Hive, the different storage formats are practically equivalent in performance. Still, the inferiority of ORC is evident. It is especially pronounced with the data points corresponding to the BED join, which are visible in Figure 8.6 as the clearly slower set of points for each worker node count. This shows that ORC seems to be simply poorly optimized all around, as it was the slowest both for reading (all queries in Figures 8.4 and 8.5) and writing (most queries in Figures 8.6 and 8.7), and it used noticeably more space to store the data set than the other formats (referring to Table 8.1).

A closer look at speedups

Excluding Impala's RCFile issue, each framework demonstrated impeccable speedups on the relatively long-running `bam` table queries. Even for Impala, this can be seen in Figure 8.3 with Parquet, though most of the queries were extremely fast with Parquet to begin with. For Hive, this is clearly visible in Figures 8.4 and 8.5. Shark provided equally obvious successes in this respect, as displayed in Figures 8.9 and 8.10.

With the small `results` table and the resultingly quite fast queries, the speedups following the BED join were not as impressive. Many improvements still took place, but the timings tended to plateau with already

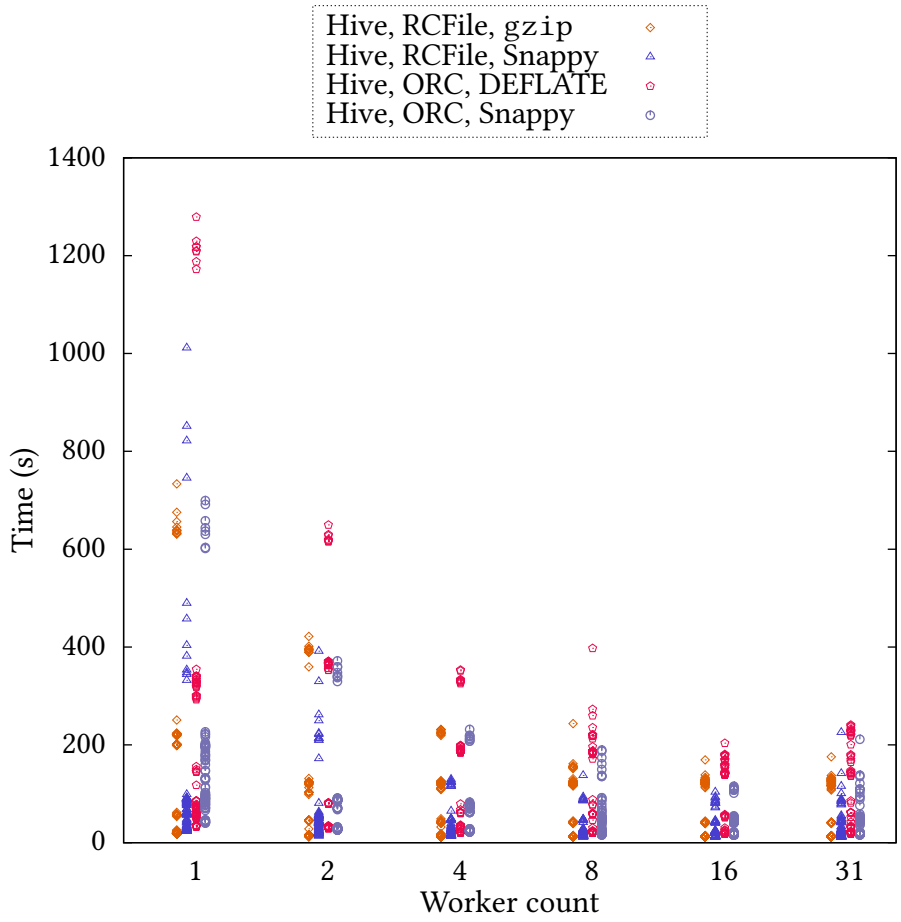

Figure 8.8: The Hive runtimes of the queries following the BED join. The data is shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `results` table.

a few worker nodes. For Hive, these results are best visible in Figure 8.8, which contains the same data points as Figures 8.6 and 8.7, but excluding the BED join times. After four or more slaves were used, most of the results no longer changed significantly. At 16 workers, the previously noticeably slower queries fell into the sub-four-minute range along with the rest, and at 31 workers, the timings tended to only worsen.

Shark achieved speedups very similar to Hive in the queries following the BED join, as shown in Figure 8.11. The main difference is that in Shark's case most of the speedup was achieved already at the two-worker mark, owing to the slowness of the very first query (counting the size of the

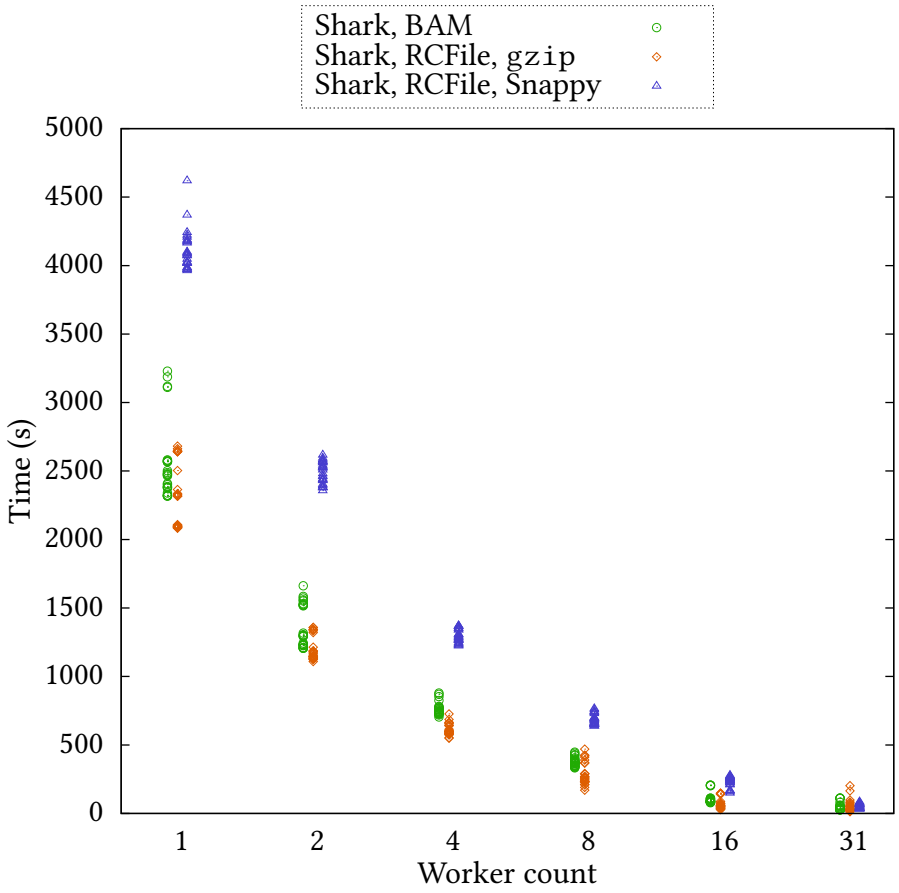

Figure 8.9: The runtime of each query run by Shark on the `bam` table. The data is shown on a linear scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `bam` table.

BED join) when only one slave node was available. As the number of worker nodes was increased, Shark demonstrated mostly gradual speedups throughout, with the biggest gap once again between 8 and 16 workers, and the difference between 16 and 31 being negligible. (Recall that all these timings may be slightly perturbed by the fact that they were timed separately using Hive's BED join output.)

Impala was extremely solid with the short `results` table queries. Their runtimes in Impala are displayed in Figure 8.12. The queries sped up cleanly up to 15 worker nodes. Most were even faster with 31 worker nodes, making Impala the only framework that managed to scale to the largest tested cluster size even with a very small amount of data. However, the `JOIN`

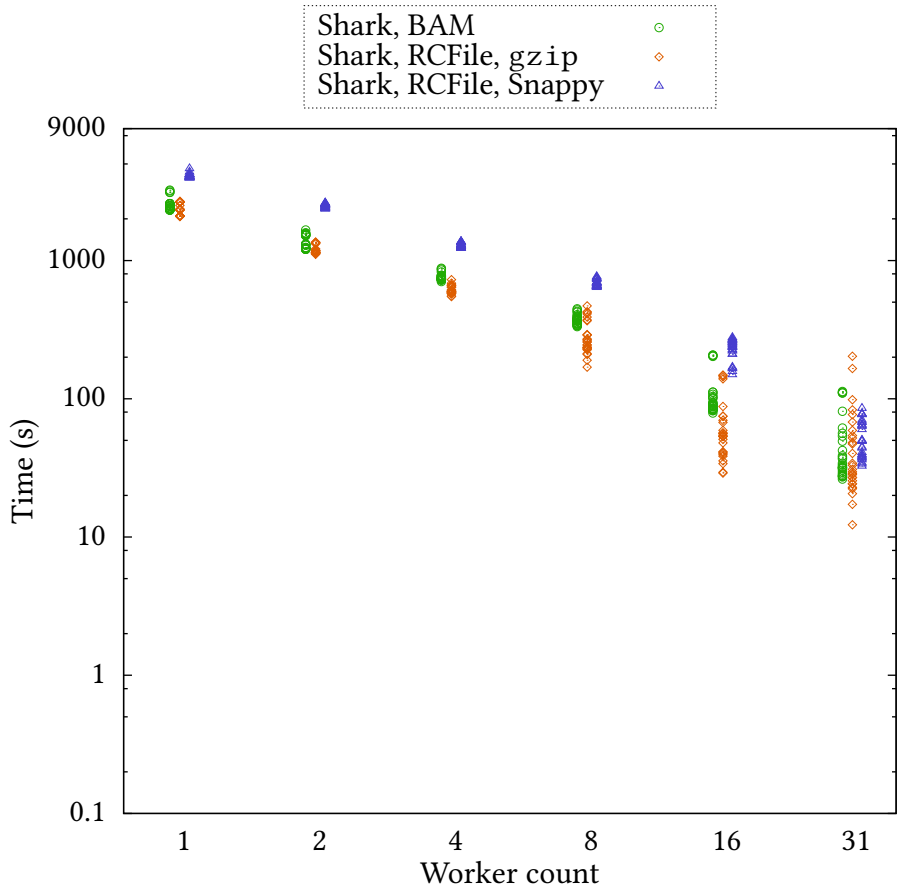

Figure 8.10: The runtime of each query run by Shark on the `bam` table. The data is shown on a logarithmic scale compared to the amount of worker nodes available, with the data points separated by the storage format of the `bam` table.

query selecting 'high-quality' reads mysteriously slowed down between 15 and 31 worker nodes, falling to a speed only slightly faster than when four slaves were used. This suggests that the benchmarked version of Impala suffers from an inefficiency in its join implementation which may only become evident in sufficiently large clusters.

Detailed comparisons

A comparison between `gzip`-compressed and Snappy-compressed RCFile that goes into more detail than the Figures seen thus far is shown in Table 8.2. The numbers in the Table are ratios between the median runtimes of the cor-

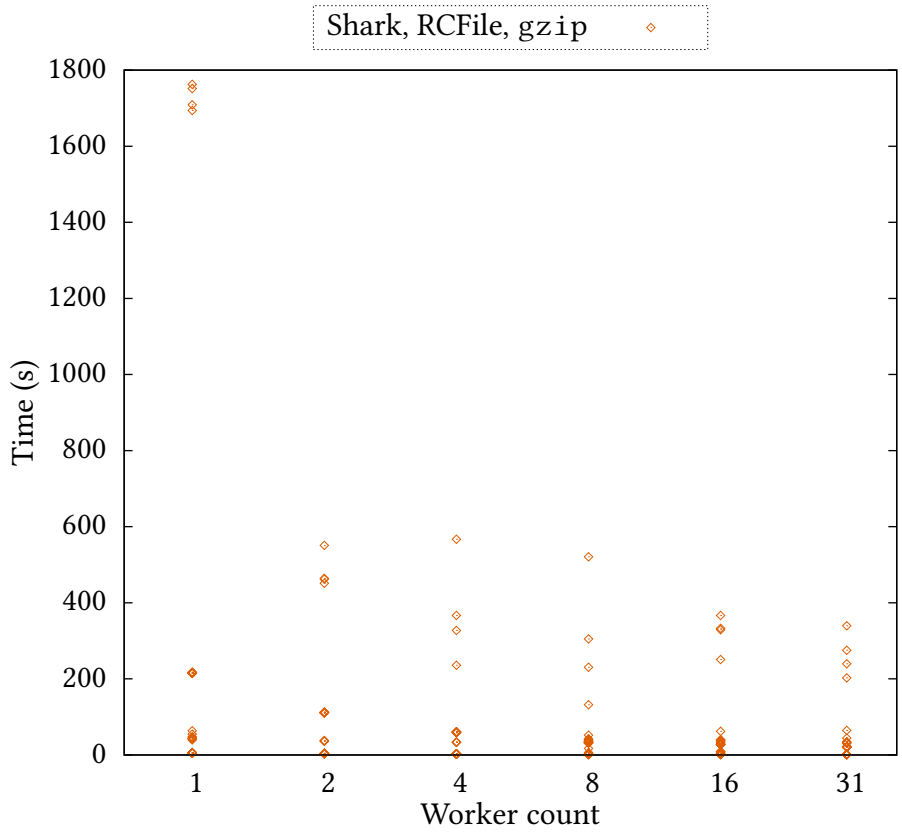

Figure 8.11: The Shark runtimes of the queries following the BED join. The data is shown on a linear scale compared to the amount of worker nodes available. The `results` table was always stored in `gzip`-compressed RCFile format. Recall that these timings were conducted separately, with the BED join result of Hive.

responding Hive-run query in each storage format and with the respective worker node count. All ratios are shown as percentages: values below 100 indicate that the query ran faster with `gzip` than with Snappy compression.

Table 8.2 demonstrates how much the type of workload can affect the optimal compressor choice. The queries operating solely on the `bam` table were significantly faster with `gzip` than with Snappy. This can be attributed to their read-only nature coupled with the previously mentioned relative excess of CPU power in the worker nodes: decompressing `gzip` was sufficiently fast that having to wait less for the local disks to provide the data set made all the difference. Referring to Table 8.1, it can be calculated

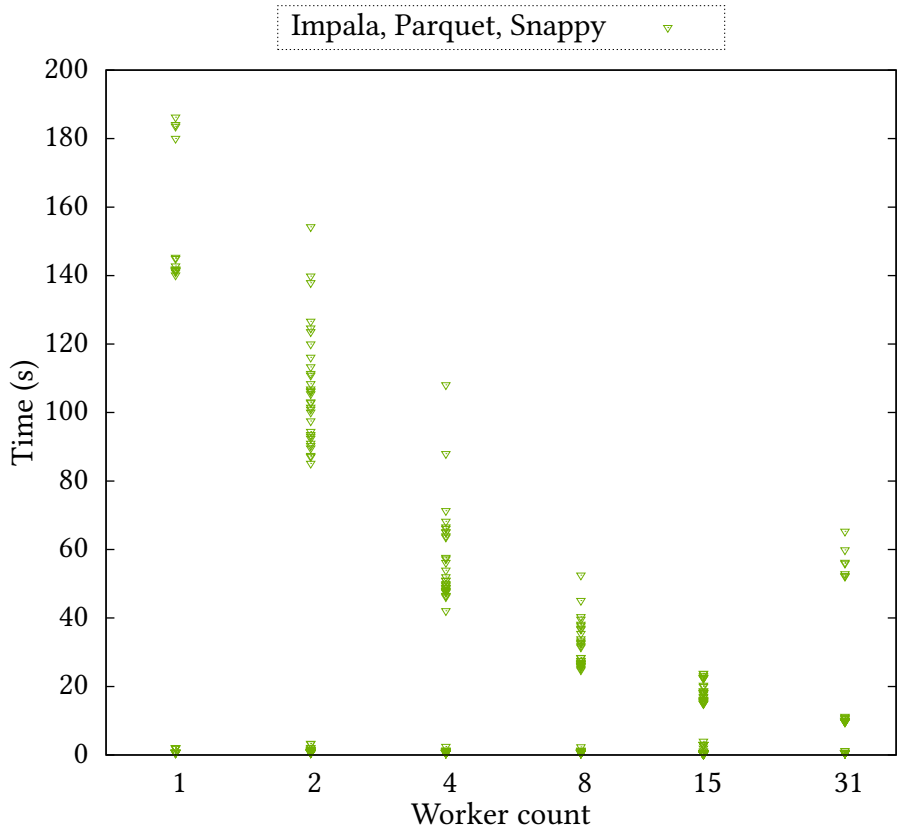

Figure 8.12: The Impala runtimes of the queries after the BED join. The data is shown on a linear scale compared to the amount of worker nodes available. The `results` table was always stored in Snappy-compressed Parquet format. Recall that the single-worker results were run separately, starting from Hive's BED join output.

that the size of the `gzip`-compressed data set was about 52.2% of the size of the Snappy-compressed data set, matching very closely with the mean speedup of `gzip` over Snappy in the queries on the full data set. With 8 and 16 worker nodes even noticeably higher speedups were achieved. One possible explanation for this improvement is that eight slave nodes was the earliest point at which the `gzip`-compressed data set could fit fully in the nodes' collective memory. Then, with 31 workers, the greater amount of disk drives relative to the data set size allowed Snappy to catch up.

Similar behaviour can be seen in the queries following the BED join: those that did not write to the `results` table—the counts and the joint mean

Workers	1	2	4	8	16	31	*mean*
	Median runtime ratio (%)						
Count all	54	46	50	41	31	87	51
Count mapped	55	47	52	34	39	82	51
Count QC-passed	55	47	52	37	37	86	52
Count unique	54	47	51	34	31	80	49
`rname` histogram	55	48	52	37	34	89	52
`mapq` histogram	54	48	53	36	36	81	51
BED join	102	107	102	98	116	137	112
Count remaining	26	64	83	67	93	85	70
`mapq` join	148	182	187	181	165	145	168
Count remaining	25	45	93	75	86	86	68
`mapq` filter	330	259	445	477	523	501	423
Count remaining	69	74	93	86	86	86	82
10-length filter	364	287	461	482	590	548	455
Count remaining	69	70	75	86	86	92	80
50-length filter	376	290	493	539	594	616	485
Count remaining	64	70	93	92	86	86	82
Mean and stddev	63	84	87	89	93	91	85

Table 8.2: Ratios between the median runtimes of each query with the `gzip`-compressed and Snappy-compressed RCFile formats, in Hive. Each number is a percentage, so values below 100 indicate that using `gzip` was faster.

and standard deviation query—were faster with `gzip`, while those that did write to it were faster with Snappy. As the data set here was small enough to fit in the main memory of even one worker node even when uncompressed, the comparison is dominated by the speeds of the compressors. The speed advantage of Snappy over `gzip` is clearly evident in the results for the three simple filtering statements, with six-fold speedups being reached in many cases. This matches with what Snappy's own documentation states about its speed: 'compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs' [Sna].

For the two join statements, the results displayed in Table 8.2 are more equal. Snappy held a distinct advantage with the reduced data set of the second join, but sometimes even fell slightly behind `gzip` in the BED join. Observing Hive's runtime strategy explains these results: Hive used two MapReduce jobs to compute the result of each of the two joins, thus

writing significant amounts of temporary data to disk in order to facilitate communication between the jobs. Therefore the joins involved both much writing and much reading. In the case of the BED join, the overwhelming difference in the size of the `bam` table between the two compressors means that despite its superiority in writes, Snappy could mostly only barely keep up with `gzip` due to causing so much more data to be read from disk. Naturally, as the amount of disk drives was increased, the size advantage of `gzip` diminished and Snappy pulled ahead. In the later join, this was realized from the start, as the data set was quite small to begin with, and so Snappy was clearly in the lead throughout. In addition, while this result has not been tabulated, it was found that when performing the BED join with differently compressed `bam` and `results` tables, the timings ranked as one would expect from Table 8.2: reading `gzip` and writing Snappy gave rise to the predominantly best speeds.

Table 8.3 provides a detailed comparison of Hive runtimes in the BAM and `gzip`-compressed RCFile formats. Since no write support was available for BAM, only the queries operating on the `bam` table are included, and the output format for the BED join was `gzip`-compressed RCFile in both cases. Values below 100 indicate that using BAM was faster than using RCFile.

Workers	1	2	4	8	16	31	*mean*
	Median runtime ratio (%)						
Count all	130	132	149	190	154	93	141
Count mapped	125	125	143	195	130	108	138
Count QC-passed	125	125	144	193	126	104	136
Count unique	125	125	145	212	126	112	141
`rname` histogram	127	126	144	199	126	86	135
`mapq` histogram	120	121	138	202	116	91	131
BED join	119	119	116	116	98	86	109

Table 8.3: Ratios between the median runtimes of each query with the BAM and `gzip`-compressed RCFile formats, in Hive. For the BED join, the output format was `gzip`-compressed RCFile. Each number is a percentage, so values below 100 indicate that using BAM was faster.

It is clear from Table 8.3 that BAM was almost always at a disadvantage to `gzip`-compressed RCFile. However, as the number of worker nodes was increased, the performance of BAM approached and sometimes even surpassed that of the RCFile format. The differences were small, and mostly

Workers	1	2	4	8	16	31	*mean*
	Median runtime ratio (%)						
Count all	49	60	59	48	20	33	45
Count mapped	60	66	70	43	21	24	47
Count QC-passed	61	66	72	48	21	26	49
Count unique	61	66	74	45	22	27	49
`rname` histogram	60	65	74	52	26	40	53
`mapq` histogram	62	68	75	52	29	33	53
BED join	86	91	91	91	83	79	87
Count remaining	31	50	54	52	65	67	53
`mapq` join	52	63	66	70	77	83	69
Count remaining	33	43	50	55	60	67	51
`mapq` filter	61	28	57	63	84	72	61
Count remaining	28	47	52	60	63	55	51
10-length filter	67	34	65	67	85	56	62
Count remaining	33	47	46	60	67	55	51
50-length filter	68	34	65	61	82	54	61
Count remaining	29	45	52	60	67	55	51
Mean and stddev	39	57	64	69	77	64	62

Table 8.4: Ratios between the median runtimes of each query with the `gzip`-compressed RCFile and DEFLATE-compressed ORC formats, in Hive. Each number is a percentage, so values below 100 indicate that using RCFile was faster.

due to the RCFile runs hitting some sort of limit between 16 and 31 worker nodes: BAM managed to speed up a bit more than RCFile at that stage. Unfortunately SequenceFiles were not included in the benchmarks—a comparison to them may have clarified whether this kind of trend is a feature of BAM or of row-oriented formats in general. Regardless, the 16-slave performance of `gzip`-compressed RCFile was so close to the 31-slave performances of both formats that it is fair to say that RCFile was the overall winner here, even if BAM managed to eke out slightly faster runtimes in the end.

In order to examine just how badly ORC performs compared to RCFile, Table 8.4 compares `gzip`-compressed RCFile to DEFLATE-compressed ORC and Table 8.5 compares Snappy-compressed RCFile to Snappy-compressed ORC. As previously, the numbers are based on the query runtimes of Hive. In both Tables, values below 100 signify that using RCFile was faster.

Workers	1	2	4	8	16	31	*mean*
	Median runtime ratio (%)						
Count all	68	86	82	73	57	29	66
Count mapped	73	90	83	82	42	27	66
Count QC-passed	73	90	85	84	45	29	68
Count unique	74	90	84	83	50	30	69
`rname` histogram	72	89	86	84	54	39	71
`mapq` histogram	76	93	85	84	57	41	73
BED join	82	87	92	83	74	57	79
Count remaining	80	90	70	75	78	83	79
`mapq` join	67	63	57	50	74	66	63
Count remaining	79	103	63	46	83	78	75
`mapq` filter	29	44	32	38	48	51	40
Count remaining	34	68	59	61	78	74	62
10-length filter	31	49	34	50	45	48	43
Count remaining	33	72	67	54	83	72	64
50-length filter	30	49	35	40	43	42	40
Count remaining	36	77	59	62	83	74	65
Mean and stddev	50	77	76	87	86	83	77

Table 8.5: Ratios between the median runtimes of each query with the RCFile and ORC formats in Hive, using Snappy compression with both formats. Each number is a percentage, so values below 100 indicate that using RCFile was faster.

The contents of Tables 8.4 and 8.5 once again clearly show that ORC was slower than RCFile in all operations. Per-query trends are not completely obvious, but there seem to be two overall progressions: with the pre-join queries concerning the `bam` table, ORC tended to slow down relatively to RCFile as more workers are added, whereas with the post-join queries operating only on the `results` table, ORC instead tended to catch up to RCFile. The latter can likely be explained with the limited room for speedup available to RCFile, given Hive's MapReduce-imposed ten-second lower bound on query runtimes. Many of the queries complete in times very near to that limit with as few as two worker nodes with RCFile, while ORC lags at least another ten seconds behind. It is unsurprising, then, that the gaps between the two formats' timings tended to narrow with the small data set.

With the `bam` table, the greatest speedups seemed to be realized near the worker counts at which the data set could fit into the collective main

memory of the nodes in only one format, as was the case when comparing `gzip` to Snappy in Table 8.2. Here in Tables 8.4 and 8.5, however, that is not a sufficient explanation, as the speedups continued to improve even when the ORC data set should also have fit entirely into memory. With the DEFLATE compressors, ORC at least catches up somewhat at the 31-worker mark, but otherwise it only slows down compared to RCFile as more nodes are available. All these ORC-related results suggest some kind of fundamental implementation issue: such poor performance is unnatural.

In Table 8.6, the runtimes of Impala and Shark, with both using the `gzip`-compressed RCFile format, are compared. Since Shark could only output `gzip`-compressed RCFile and Impala could only output Snappy-compressed Parquet, only the queries depending on the input-only `bam` table, i.e. those prior to and including the BED join, are included. Values below 100 denote that Impala was faster.

Workers	1	2	4	8	16*	31	*mean*
	Median runtime ratio (%)						
Count all	70	80	426	1011	1595	1426	772
Count mapped	70	78	493	389	3556	3886	1373
Count QC-passed	70	80	499	401	2033	4319	1264
Count unique	70	78	636	318	1892	4993	1328
`rname` histogram	68	76	93	33	156	175	109
`mapq` histogram	69	76	102	60	185	1141	314
BED join	*N/A*†	116	114	123	131	333	216

Table 8.6: Ratios between the median runtimes of the queries on the `bam` table in Impala and Shark, both with the `gzip`-compressed RCFile format. Each number is a percentage, so values below 100 indicate that Impala was faster.
*For Impala, 15 workers were used instead of 16.
†Impala was unable to run the `JOIN` statement with only a single worker.

Table 8.6 illustrates Impala's RCFile issues once again: with only one or two worker nodes, Impala was solidly faster than Shark. However, beyond that point, the performance numbers fluctuated erratically. Even using the medians to somewhat curb the variance does not hide this behaviour, and for most queries the performance of Impala was simply poor with more than two slaves. The more complex queries seemed to behave better, but all suffered dramatic slowdowns by the 15 or 31 worker node marks.

The only results in Table 8.6 that seem to have no unexpected issues are the ones where the runs were performed with one or two workers. Unfortunately it makes little sense to discuss trends with only two data points per query: e.g. while it seems that Shark slightly caught up to Impala when the cluster size was increased, it is impossible to determine whether such behaviour would have continued without Impala's RCFile issue. If Impala's best-case results with eight worker nodes would have been consistent, that trend would have reversed for at least some of the queries. The BED join results are also, despite their greater stability, able to clarify little, since the fact that Shark computed an incorrect result makes its timings barely comparable to Impala's. Impala's inability to compute the join in the case of one worker node also complicates the analysis.

To approximate how Impala's performance might compare to Shark if the results were not plagued by the mysterious RCFile bug, Table 8.7 displays the simple case where Impala uses Snappy-compressed Parquet throughout and Shark uses `gzip`-compressed RCFile throughout. Of course, the Table may most aptly constitute a comparison of Parquet to RCFile: based on the available data it is impossible to know how much of the relative performance is due to the frameworks, as opposed to the different file formats. As such, the precise values in the Table are not as meaningful as in the previous comparisons. One can still, though, observe the differences between the values to see how different queries or worker node counts affect the relative runtimes.

The BED join numbers in Table 8.7 are mostly meaningless, as they are in Table 8.6. Shark computed an incorrect zero-length result and so it is hardly surprising that it was faster than even the otherwise clearly victorious Impala. In addition, Impala could not perform the join at all with only one worker node.

The `bam` table queries completed ridiculously fast with Parquet, as also seen previously in Figure 8.3. Even with only one worker node, Impala finished each one in less than a minute. Table 8.7 re-iterates just how great a difference this is: Impala was often about 200 times as fast as Shark. Parquet's extensive built-in metadata clearly allow the kinds of simple filters and groupings represented by the queries on the `bam` table to be completed without requiring more than a small fraction of the whole data set: fully scanning the 452.4 TiB data set in one minute would have required nearly 65 Gbps of throughput, a value well beyond even the theoretical maximum of the network used.

Table 8.7 shows that in general, Shark sped up much more between the one and two worker node marks than did Impala, as was also visible in Table 8.6. The only cases where this trend was not present were the `bam`

Workers	1	2	4	8	16*	31	*mean*
	Median runtime ratio (%)						
Count all	2.1	1.3	4.0	5.1	13.1	7.2	5.5
Count mapped	2.3	1.6	3.3	3.6	15.0	6.4	5.4
Count QC-passed	0.5	0.6	0.7	0.9	2.6	3.3	1.4
Count unique	0.5	0.6	0.7	0.9	2.3	3.4	1.4
`rname` histogram	2.4	1.9	1.9	2.3	8.3	4.8	3.6
`mapq` histogram	1.3	1.3	1.4	2.3	6.5	4.5	2.9
BED join	*N/A*†	146.3	208.8	269.3	259.8	115.0	199.8
Count remaining	11.7	18.3	18.1	24.8	32.6	34.4	23.3
`mapq` join	10.6	28.6	19.1	15.5	6.8	20.3	16.8
Count remaining	1.4	19.9	24.8	34.3	13.4	54.8	24.8
`mapq` filter	66.4	90.5	80.8	84.0	62.1	48.5	72.1
Count remaining	12.1	20.9	24.9	25.1	31.9	54.1	28.2
10-length filter	65.6	91.7	82.2	74.0	48.4	35.7	66.3
Count remaining	14.1	21.2	30.6	44.4	48.2	57.0	35.9
50-length filter	65.9	79.9	82.9	80.4	63.5	51.6	70.7
Count remaining	14.7	23.6	24.9	33.4	17.2	59.3	28.9
Mean and stddev	4.7	5.0	3.1	2.3	1.7	1.7	3.1

Table 8.7: Ratios between the median runtimes of each query in Impala and Shark, using Snappy-compressed Parquet and `gzip`-compressed RCFile respectively. Each number is a percentage, so values below 100 indicate that Impala was faster. Since quite small numbers are involved, an additional decimal is used.
*For Impala, 15 workers were used instead of 16.
†Impala was unable to run the `JOIN` statement with only a single worker. Thus the single-worker timings for the later queries were obtained separately, using Hive's join result.

table queries with Parquet. Of course this may be due to how much room for speedup there was: the queries run on `results` were already initially so fast in Impala that keeping such a lead on Shark may be a completely unrealistic proposition.

Lastly, Table 8.7 shows the power of Impala's LLVM-based code generation. While Impala was typically 2–4 times as fast as Shark in the `results` table queries, the last query run, which computes the mean and standard deviation, was completed *20–50* times as quickly by Impala as by Shark. Clearly, as soon as some nontrivial arithmetic is involved, Impala is far ahead of its competition.

Chapter 9

Conclusions

> The most merciful thing in the world, I think, is the inability of the human mind to correlate all its contents.
>
> Francis Wayland Thurston
> 'The Call of Cthulhu'
> H. P. Lovecraft [Lov28]

This Thesis discussed interactive analysis of sequencing data using SQL-based frameworks designed for working with Big Data. The results of the experiments show that the emergence of frameworks made with interactivity in mind is a boon for exploratory analysts. The difference in response time between Hive, which lacks such considerations, and both Shark and Impala is, in the best case, immense. The ten-second lower bound on the runtime of MapReduce jobs [Pav+09; Xin+12] makes Hive practically unsuitable for interactive use. Furthermore, storing one's data set in an appropriate storage format, designed to have these frameworks operate at their highest effectiveness, can make an even greater difference to just using Hive with a 'default' format such as BAM. New advanced columnar formats such as Parquet are especially promising in this regard.

Regrettably, there was not enough time to investigate the effect of various features that may have tipped the scales in Hive's favour. For example, Hive's indexes are utilized by neither Shark [ShC] nor Impala [ImQ], and therefore might have noticeably narrowed the performance gap between Hive and the others. In addition, while table and column statistics are supported by both Hive and Impala and while partitioned tables are supported by all three frameworks benchmarked, the improvements resulting from these may apply more significantly to Hive than the others, since

Hive's longer runtimes give it more room for improvement. Thus, it may be possible to create a setup in which Hive performs far more reasonably for interactive use than in these experiments. However, the ten-second lower bound on MapReduce job runtime remains unconquerable with modifications such as these. In order for Hive to be able to compete with Shark and Impala, the use of MapReduce must be limited by turning to alternative computational backends such as Tez [Mur+13; TeH; Tez].

All benchmarked frameworks sped up very well on long-running queries on the large data set all the way up to the largest cluster size tested, with 31 worker nodes. When it came to more interactive queries, however, improvements after adding more than a few worker nodes were scant. In part this is understandable and acceptable, as runtimes approaching the single-second mark are at the point where noticeable improvements are both difficult to achieve and not particularly necessary. In the case of Hive, it is understandable due to the previously mentioned limitations of MapReduce, but still makes for a disappointing comparison with Shark and especially Impala, both of which were able to push runtimes much lower than that.

Unimplemented features and bugs in the tested versions of the frameworks (Hive 0.11, Shark 0.7, and Impala 1.0.1) prevented many interesting benchmarks from being performed. While several have been fixed in new releases made since then [HIV10; ImB13; ImF; SHA13], others remain unattended to [ImL; ImQ; ImU; SHA12]. Particularly Shark and Impala exhibited many limitations and caused many re-runs due to crashes. As such it must be concluded that if one is not prepared to deal with any unexpected issues, one should stick to the very stable Hive framework.

The BAM format was, for the most part, not the optimal choice, as was to be expected. The columnar formats (apart from ORC as discussed below), with their optimizations for distributed processing, naturally perform better at what they were designed for. While it seems to be possible to use a large enough cluster to overcome the limitations of BAM, that is simply inefficient use of resources. The resulting wins are hardly worth the expenditure: a columnar format can perform almost as well with far fewer nodes. Unfortunately time constraints prevented a comparison to SequenceFiles that would have clarified whether other row-oriented formats perform similarly to BAM. In all, it seems that despite its own optimizations, BAM is not the best choice for Big Data sets of sequencing data, regardless of use case. For solely archival purposes, even the simple alternative of `gzip`-compressed SAM would be better, as the resulting file size tends to be smaller—not only is compressed SAM slightly smaller than BAM, but `gzip` gives about 8% smaller compressed sizes than BGZF. Whether this translates to also mak-

ing SAM preferable for computational purposes remains unknown, so BAM can be thought of as a reasonable trade-off. Nevertheless, both RCFile and Parquet performed better on average, and a standardized file format based on either of these would be a viable and more efficient replacement for BAM.

Among the columnar formats, RCFile had the smallest size out of both the DEFLATE-compressed and Snappy-compressed files. It also offered respectable runtimes for the queries benchmarked, making it a very good choice for any use case.

Impala's results showed, however, that Parquet is far ahead of RCFile in query performance. Of course the specific queries discussed will affect this conclusion somewhat, but Parquet is well optimized for several common types of filters and statistics and will likely continue to perform better than RCFile in any real-world workload. The issue affecting Impala's RCFile performance, while somewhat muddying the results, is unlikely to have affected the trends so much as to weaken this conclusion. The difference in data set size between the Snappy-compressed RCFile and Snappy-compressed Parquet formats suggests that Parquet's size will be at worst only slightly larger than that of RCFile regardless of the compressor. Overall, Parquet seems to be the best choice for any use case except those in which data set size is of paramount importance.

ORC performed extremely poorly compared to both RCFile and Parquet. The size of the data set in ORC was the largest among all the formats, and query performance was even slower than could be explained by the size disparity alone. Perhaps future advancements in the Stinger Initiative [StI] will make ORC more competitive with the other formats. Until then, based on the results obtained here, ORC cannot be recommended for sequencing data—a conclusion that unexpectedly contradicts previous comparisons of ORC to RCFile [OMa13].

Aside from the file formats, the choice of compressor was also found to be an important factor in query performance. Naturally this was the case with the large data set, in which the difference between the DEFLATE-compressed and the Snappy-compressed size was over 200 GiB, but the choice was found to have significant effects even on queries run on the much smaller reduced data set. In a hardware environment more suited for Hadoop (constructible e.g. by following the Cloudera recommendations [ODe13]) than the CPU-heavy cluster used for benchmarking in this Thesis, the query runtimes would likely have been even more strongly affected, potentially giving an edge to Snappy.

All in all, it appears that with modern tools and reasonably large clusters, interactive exploratory analysis of Big sequencing data is currently vi-

able. Of course more complicated analyses, involving far costlier operations than benchmarked here, such as joining multiple large BAM files together, are also possible and of interest to bioinformaticians. For such workloads, running at interactive speeds is likely to require optimizing the existing solutions further, if not creating entirely new solutions. Thankfully, there is plenty of room for improvement in all of Hive, Shark, and Impala as well as in the newer file formats, Parquet and ORC, and all are in active development. Other solutions based on SQL-on-Hadoop which were not evaluated in this work, such as Presto [Pre; Tra13], Tez, and the unconventional BlinkDB [Aga+12; Aga+13; BDB], are also promising. In addition, the fault tolerance issue in large clusters appears to have been kept well in mind, with nearly all frameworks having a solution of some kind. The future seems to bode well for all kinds of interactive Big Data computations.

On the other hand, Big Data is only growing bigger. Whether this proves to be an insurmountable issue regarding interactive analysis remains to be seen. For now, assuming that at least one of the tested frameworks can continue scaling to more nodes as effectively as in these experiments, warehouse-scale computers should be able to handle any current sequencing data set. There are now a large number of SQL-on-Hadoop-based query engines designed for interactive analysis of Big Data sets: it remains to be seen which one of them will see widespread adoption.

Appendix A

Experimental configuration

A short listing of the most important settings used during the experiments presented in Chapter 7 was presented in Section 7.5. In this Appendix, almost every individual setting is shown, including ones set in configuration files, those given as command line parameters, and relevant environment variables. The only settings not listed here are those specifying highly environment-specific information such as network host names and ports, paths to software components and configuration directories, usernames and passwords, etc.

Since the XML (Extensible Markup Language) [Bra+08] format used for much of the configuration is rather verbose, the settings are not displayed in their precise original format, but instead as simple name-value pairs. Still, some names are long enough to require line wrapping. Instead of hyphenation, which could cause confusion regarding whether the hyphen is part of the name or not, such line wraps are indicated with an ellipsis after the line break.

A.1 Hadoop

Tables A.1 and A.2 list the settings from two main Hadoop XML files: `core-site.xml` and `hdfs-site.xml`, respectively.

For security reasons, `dfs.domain.socket.path`, shown in Table A.2, was not allowed to be on the local file system nor in the global temporary data directory (`/tmp`)—those choices were rejected programmatically. The only alternatives were the user's home directory, which resided on an NFS (Network File System), or the user's private directory on the Lustre file system. While the end result of placing a domain socket on a network file

Name	Value
io.sort.mb	1280
fs.inmemory.size.mb	1280
io.sort.factor	100
io.file.buffer.size	65536
hadoop.tmp.dir	a directory on the local file system
io.compression.codecs	added org.apache.hadoop.io.compress.SnappyCodec

Table A.1: Hadoop settings given in core-site.xml.

Name	Value
dfs.client.file-block-storage-locations.timeout	3000
dfs.client.read.shortcircuit	true
dfs.datanode.hdfs-blocks-metadata.enabled	true
dfs.namenode.handler.count	32
dfs.replication	max(3, $\lfloor n/2 \rfloor$) where n was the number of slave nodes
dfs.domain.socket.path	a directory on the Lustre file system
dfs.data.dir	a directory on the local file system

Table A.2: HDFS settings given in hdfs-site.xml.

system may seem curious, the performance impact should be negligible as the file system itself is not involved when communicating over the socket.

The remaining Hadoop configuration file settings are shown in Table A.3, while Table A.4 lists the used Hadoop-affecting environment variables.

Additionally, the command ulimit -u 4096 was used to increase the maximum number of running processes allowed for the user. This was required for Impala, as it sometimes caused the HDFS datanode processes to reach the default limit of 1024.

Name	Value
mapred.tasktracker.map. ···tasks.maximum	12
mapred.tasktracker. ···reduce.tasks.maximum	6
mapred.job.tracker. ···handler.count	32
mapred.child.java.opts	-Xmx3072m
mapred.compress.map.out-put	true
mapred.map.output. ···compression.codec	org.apache.hadoop.io. ···compress.SnappyCodec

Table A.3: Hadoop MapReduce settings given in `mapred-site.xml`.

Name	Value
HADOOP_HEAPSIZE	2048
HADOOP_NICENESS	0
HADOOP_PID_DIR	a directory on the local file system
HADOOP_LOG_DIR	a directory on the Lustre file system

Table A.4: The relevant environment variables for Hadoop.

A.2 Hive

Table A.5 lists the settings used for Hive, most of which were given in the main `hive-site.xml` configuration file.

The Hive frontend processes were started with the `-v` option for additional verbosity.

A.3 Shark

Environment variables relevant to Shark are shown in Table A.6.

The `SPARK_JAVA_OPTS` environment variable was modified to include the settings displayed in Table A.7 in addition to the defaults from the provided `shark-env.sh.template` file.

All Spark workers were started with the `-m 44G` command line parameter, giving some extra memory for each worker as a whole beyond the

Name	Value
hive.exec.parallel	true
hive.exec.compress.intermediate	true
hive.merge.mapredfiles	true
hive.stats.ndv.error	5.0
hive.exec.scratchdir	a directory on the local file system
hive.querylog.location	a directory on the Lustre file system
hive.log.dir	a directory on the Lustre file system

Table A.5: Hive configuration variables.

Name	Value
SHARK_MASTER_MEM	2g
SPARK_MEM	40G

Table A.6: Shark environment variables.

Name	Value
spark.local.dir	a directory on the local file system
spark.worker.timeout	30000
spark.akka.timeout	30000
spark.storage.blockManagerHeartBeatMs	30000
spark.akka.retry.wait	30000
spark.akka.frameSize	10000

Table A.7: Parameters given in SPARK_JAVA_OPTS.

per-task SPARK_MEM setting. The -d option was used to specify a scratch space directory on the local file system.

For additional verbosity, -v was passed to the Shark frontend process.

A.4 Impala

The ulimit -u 4096 command was used before starting the impalad processes, for reasons similar to its inclusion in the Hadoop settings.

Table A.8 shows the environment variable settings for Impala. Note that they all concern only logging.

Name	Value
`IMPALA_LOG_DIR`	a directory on the Lustre file system
`GLOG_v`	1
`GLOG_minloglevel`	0
`GLOG_stderrthreshold`	0
`GLOG_logtostderr`	1

Table A.8: Impala environment variables, all concerning only logging.

As with Hive and Shark, additional verbosity was requested from the Impala frontend process with the appropriate command line option, `-v`. In addition, the `-r` option was used in place of the `refresh` command.

Appendix B

HiveQL statements used

The HiveQL statements which were used in the experiments presented in Chapter 7 are listed here. They are split up into three Sections in accordance with the three groups explained in Section 7.4.

Each HiveQL Listing is accompanied by a marginal note demarcating the frameworks with which the program code in the Listing was used. Only the first letter of each framework's name is used: H for Hive, S for Shark, and I for Impala. Different code for different frameworks may be included in one Listing, in which case the alternatives are separated with dotted lines. Below is an example Listing.

```
Example Hive-only code.                                          H
.....................................................................
Example code used with Shark and Impala.                         SI
```

B.1 Table creation and settings

Table creation statements are shown only for the `bam` table. For all storage formats used, the `results` table was always created identically to the `bam` table in the same format, with only the name of the table differing. Furthermore, the statements for each combination of input and output formats are not listed in full: instead, only the individual statements from which the combinations can trivially be reconstructed are shown.

The table schema was always the same, as shown in Listing B.1 below.

Listing B.1: HiveQL code describing the table schema.

```
qname STRING, flag SMALLINT, rname STRING, pos INT,              HSI
mapq TINYINT, cigar STRING, rnext STRING, pnext INT,
```

```
tlen INT, seq STRING, qual STRING, opts_char STRING,
opts_int STRING, opts_float STRING, opts_string STRING,
opts_arr_int8 STRING, opts_arr_int16 STRING,
opts_arr_int32 STRING, opts_arr_float STRING
```

For the sake of brevity, this set of columns is not repeated below. In its stead, the marker *columns* is used.

Listing B.2: The HiveQL statement used to create the `bam` table in BAM format.

```
CREATE TABLE bam ROW FORMAT SERDE "SAMSerDe" STORED AS
    INPUTFORMAT "DeprecatedBAMInputFormat"
    OUTPUTFORMAT "HiveKeyIgnoringBAMOutputFormat";
```

HS

Note that due to the usage of a SerDe, the columns of Listing B.1 were not given in the statement shown in Listing B.2: the SerDe itself defines the schema.

Listing B.3: The HiveQL statement creating the `bam` table in RCFile format.

```
CREATE TABLE bam (columns) STORED AS RCFILE;
```

HSI

Listing B.4: HiveQL statements creating the `bam` table in ORC format, showing both DEFLATE and Snappy compression.

```
CREATE TABLE bam (columns) STORED AS ORC
   TBLPROPERTIES ("orc.compress" = "ZLIB");
CREATE TABLE bam (columns) STORED AS ORC
   TBLPROPERTIES ("orc.compress" = "SNAPPY");
```

H

Listing B.5: The HiveQL statement creating the `bam` table in Parquet format.

```
CREATE TABLE bam (columns) STORED AS PARQUETFILE;
```

I

Listing B.6: The HiveQL statement used to create the BED table.

```
CREATE TABLE bed
    (chrom STRING, chromStart INT, chromEnd INT)
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;
```

HSI

Listing B.7: The RCFile compression settings used with both compressors.

```
SET hive.exec.compress.output=true;
```

HS

```
SET mapred.max.split.size=256000000;
SET mapred.output.compression.type=BLOCK;
```

Listing B.8: The RCFile `gzip` compression setting.

```
SET mapred.output.compression.codec=
   org.apache.hadoop.io.compress.GzipCodec;
```

HS

Listing B.9: The RCFile Snappy compression setting.

```
SET mapred.output.compression.codec=
   org.apache.hadoop.io.compress.SnappyCodec;
```

H

Listing B.10: HiveQL code creating the tables and enabling the settings that were used when separately benchmarking the post-BED join exploratory queries on the `results` table in Shark.

```
CREATE TABLE orig (columns) STORED AS RCFILE;
CREATE TABLE results (columns) STORED AS RCFILE;
SET hive.exec.compress.output=true;
SET mapred.max.split.size=256000000;
SET mapred.output.compression.type=BLOCK;
SET mapred.output.compression.codec=
   org.apache.hadoop.io.compress.GzipCodec;
```

S

Listing B.11: HiveQL code creating the tables used when separately benchmarking the post-BED join exploratory queries on the `results` table in Impala with one worker node.

```
CREATE TABLE orig (columns) STORED AS PARQUETFILE;
CREATE TABLE results (columns) STORED AS PARQUETFILE;
```

I

B.2 Queries on the full data set

Listing B.12: The parallelism setting, which was in place also for the exploratory queries.

```
SET mapred.reduce.tasks = R;
```

HS

Listing B.13: The initial counting statements on the full data set.

```
SELECT COUNT(*) AS total FROM bam;
```

HSI

```
SELECT COUNT(*) AS mapped FROM bam WHERE flag & 4 = 0;
SELECT COUNT(*) AS passedQC FROM bam
   WHERE flag & 512 = 0;
SELECT COUNT(*) AS notDuplicate FROM bam
   WHERE flag & 1024 = 0;
```

Listing B.14: The statements computing the two histograms, by name and quality respectively.

```
SELECT rname, COUNT(*) FROM bam                                    HS
   GROUP BY rname ORDER BY rname;

SELECT PMOD(mapq,256) AS pmapq, COUNT(*) FROM bam
   WHERE flag & (4 | 1024) = 0
   GROUP BY mapq ORDER BY pmapq;
.....................................................................
SELECT rname, COUNT(*) FROM bam                                    I
   GROUP BY rname ORDER BY rname LIMIT 100;

SELECT PMOD(mapq,256) AS pmapq, COUNT(*) FROM bam
   WHERE flag & (4 | 1024) = 0
   GROUP BY mapq ORDER BY pmapq LIMIT 256;
```

The full list of columns in the bam table is shown in Listing B.15. For brevity, the *bamColumns* marker is used below instead of the full code.

Listing B.15: Code specifying the columns in the bam table.

```
bam.qname, bam.flag, bam.rname, bam.pos, bam.mapq,                 HSI
bam.cigar, bam.rnext, bam.pnext, bam.tlen, bam.seq,
bam.qual, bam.opts_char, bam.opts_int, bam.opts_float,
bam.opts_string, bam.opts_arr_int8, bam.opts_arr_int16,
bam.opts_arr_int32, bam.opts_arr_float
```

Listing B.16: The join with the BED table.

```
INSERT OVERWRITE TABLE results                                     H
   SELECT DISTINCT bamColumns
   FROM bed JOIN (SELECT * FROM bam WHERE
                     flag & 4 = 0 AND seq <> "*") bam
             ON bam.rname = bed.chrom
   WHERE bam.pos                     <= bed.chromEnd
     AND bam.pos + length(bam.seq) >= bed.chromStart;
```

```
.....................................................................
INSERT OVERWRITE TABLE results                                        S
   SELECT /*+ MAPJOIN(bed) */ DISTINCT bamColumns
   FROM bed JOIN (SELECT * FROM bam WHERE
                     flag & 4 = 0 AND seq <> "*") bam
              ON bam.rname = bed.chrom
   WHERE bam.pos                     <= bed.chromEnd
     AND bam.pos + length(bam.seq) >= bed.chromStart;
.....................................................................
INSERT OVERWRITE TABLE results                                        I
   SELECT DISTINCT bamColumns
   FROM (SELECT * FROM bam WHERE
            flag & 4 = 0 AND seq <> "*") bam
        JOIN bed ON bam.rname = bed.chrom
   WHERE bam.pos                     <= bed.chromEnd
     AND bam.pos + length(bam.seq) >= bed.chromStart;
```

As Shark was not able to compute the BED join correctly, the result from Hive was used instead. An initial copy of the data was made in case of any differences between Hive's and Shark's outputs, as shown in Listing B.17. The same code was used for Impala's single-worker runs due to the similar problem with the BED join therein.

Listing B.17: The statement taking Hive's separately computed BED join result into use for the later exploratory queries.

```
INSERT OVERWRITE TABLE results SELECT * FROM orig;                    SI
```

Listing B.18: The query counting the size of the result of the BED join.

```
SELECT COUNT(*) FROM results;                                         HSI
```

B.3 Exploratory queries on the reduced data set

Listing B.19: The join used to select for highest mapping quality, and the count computing the result's size.

```
INSERT OVERWRITE TABLE results                                        HS
   SELECT results.*
   FROM (SELECT pos, tlen, MAX(PMOD(mapq,256)) AS maxq
```

```
            FROM results
            WHERE flag & 4 = 0 AND mapq <> -1
            GROUP BY pos, tlen) tmp
      JOIN results
      ON     results.pos = tmp.pos
         AND results.tlen = tmp.tlen
   WHERE PMOD(results.mapq,256) = tmp.maxq;
SELECT COUNT(*) FROM results;
.....................................................................
INSERT OVERWRITE TABLE results                                        I
   SELECT results.*
   FROM results JOIN
      (SELECT pos, tlen, MAX(PMOD(mapq,256)) AS maxq
          FROM results
          WHERE flag & 4 = 0 AND mapq <> -1
          GROUP BY pos, tlen) tmp
      ON     results.pos = tmp.pos
         AND results.tlen = tmp.tlen
   WHERE PMOD(results.mapq,256) = tmp.maxq;
SELECT COUNT(*) FROM results;
```

Listing B.20: The simple filters and the interspersed counts on the `results` table.

```
INSERT OVERWRITE TABLE results                                        HSI
   SELECT * FROM results
      WHERE mapq <> -1 AND PMOD(mapq,256) >= 60;
SELECT COUNT(*) FROM results;
INSERT OVERWRITE TABLE results
   SELECT * FROM results WHERE LENGTH(seq) >= 10;
SELECT COUNT(*) FROM results;
INSERT OVERWRITE TABLE results
   SELECT * FROM results WHERE LENGTH(seq) >= 50;
SELECT COUNT(*) FROM results;
```

Listing B.21: The roundabout mean and standard deviation calculation.

```
SELECT AVG(tlen),                                                     H
   SQRT(SUM(POW(tlen - mean, 2)) / (COUNT(*) - 1)) FROM
   (SELECT AVG(tlen) AS mean FROM results) tmp
   JOIN results;
.....................................................................
```

```
SELECT AVG(tlen),                                                    S
   SQRT(SUM(POW(tlen - mean, 2)) / (COUNT(*) - 1)) FROM
   results JOIN
   (SELECT AVG(tlen) AS mean FROM results) tmp;
.....................................................................
SELECT AVG(tlen),                                                    I
   SQRT(SUM(POW(tlen - mean, 2)) / (COUNT(*) - 1)) FROM
   (SELECT tlen, tlen-tlen as x FROM results) tmp
   JOIN
   (SELECT AVG(tlen) AS mean,
           COUNT(tlen)-COUNT(tlen) as x
    FROM results) tmp2
   ON tmp.x = tmp2.x;
```

In Shark, the order of the tables in the join in Listing B.21 was found to have no measurable effect on the runtime. The ordering shown was chosen to differ from Hive's purely due to whimsy.

Bibliography

[1kG] *1000 Genomes. A Deep Catalog of Human Genetic Variation.* URL: http://www.1000genomes.org.

[Aga+12] Sameer Agarwal, Aurojit Panda, Barzan Mozafari, Anand P. Iyer, Samuel Madden, and Ion Stoica. "Blink and It's Done: Interactive Queries on Very Large Data". In: *PVLDB* 5.12 (2012), pp. 1902–1905. URL: http://vldb.org/pvldb/vol5/p1902_sameeragarwal_vldb2012.pdf.

[Aga+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 'BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data'. In: *EuroSys*. Ed. by Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek. ACM, 2013, pp. 29–42. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465355.

[Ail+01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 'Weaving Relations for Cache Performance'. In: *VLDB*. Ed. by Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass. Morgan Kaufmann, 2001, pp. 169–180. ISBN: 1-55860-804-4. URL: http://www.vldb.org/conf/2001/P169.pdf.

[Aiy+12] Amitanand S. Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. 'Storage Infrastructure Behind Facebook Messages: Using HBase at Scale'. In: *IEEE Data Engin-*

eering Bulletin 35.2 (2012), pp. 4–13. URL: http://sites.computer.org/debull/A12june/facebook.pdf.

[Ala+13] Jyrki Alakuijala and Lode Vandevenne. *Data compression using Zopfli*. Tech. rep. Google Inc., Feb. 2013. URL: https://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf.

[Ale+11] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. 'MapReduce and PACT - Comparing Data Parallel Programming Models'. In: *Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW)*. BTW 2011. Kaiserslautern, Germany: GI, 2011, pp. 25–44. ISBN: 978-3-88579-274-1. URL: http://stratosphere.eu/assets/papers/ComparingMapReduceAndPACTs_11.pdf.

[And] Simon Andrews. *FastQC: A Quality Control tool for High Throughput Sequence Data*. Babraham Bioinformatics. URL: http://www.bioinformatics.babraham.ac.uk/projects/fastqc/.

[Apa] *The Apache Software Foundation*. URL: https://www.apache.org.

[Arm97] Joe L. Armstrong. 'The Development of Erlang'. In: *ICFP*. Ed. by Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman. ACM, 1997, pp. 196–203. ISBN: 978-0-89791-918-0. DOI: 10.1145/258948.258967.

[Aur+12] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Boris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang. 'Data Infrastructure at LinkedIn'. In: *ICDE*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 1370–1381. ISBN: 978-0-7685-4747-3. DOI: 10.1109/ICDE.2012.147.

[Avr13] *Apache Avro™ 1.7.4 Specification*. Apache Software Foundation. 26th Feb. 2013. URL: https://avro.apache.org/docs/1.7.4/spec.html.

[Bac+02] David F. Bacon, Stephen J. Fink, and David Grove. 'Space- and Time-Efficient Implementation of the Java Object Model'. In: *ECOOP*. Ed. by Boris Magnusson. Vol. 2374. Lecture Notes in Computer Science. Springer, 2002, pp. 111–132. ISBN: 3-540-43759-2. DOI: 10.1007/3-540-47993-7_5.

[Bar+03] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 'Web Search for a Planet: The Google Cluster Architecture'. In: *IEEE Micro* 23.2 (2003), pp. 22–28. DOI: 10.1109/MM.2003.1196112.

[Bar+13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Second edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, July 2013. DOI: 10.2200/S00516ED2V01Y201306CAC024.

[BCF] *BCF (Binary VCF) version 2*. Version 2.1. URL: http://www.1000genomes.org/wiki/analysis/variant-call-format/bcf-binary-vcf-version-2.

[BDB] *BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data*. URL: http://blinkdb.org.

[BED13] *BED File Format*. Version 73. Sept. 2013. URL: http://www.ensembl.org/info/website/upload/bed.html.

[Ber+05] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Internet Standard). Network Working Group, Jan. 2005. URL: https://tools.ietf.org/html/rfc3986.

[Bie93] Ambrose Bierce. *The Devil's Dictionary*. Ed. by Aloysius of &tSftDotIotE. 15th Apr. 1993. URL: http://www.gutenberg.org/ebooks/972.

[Blo70] Burton H. Bloom. 'Space/Time Trade-offs in Hash Coding with Allowable Errors'. In: *Communications of the ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692.

[Bor+11] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. 'Apache Hadoop Goes Realtime at Facebook'. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 1071–1080. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989438.

[Bos13] Pradip Bose. 'Is dark silicon real?: technical perspective'. In: *Communications of the ACM* 56.2 (2013), p. 92. DOI: 10.1145/2408776.2408796.

[Bra+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, eds. *Extensible Markup Language (XML) 1.0*. 5th ed. 26th Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[Bu+10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 'HaLoop: Efficient Iterative Data Processing on Large Clusters'. In: *PVLDB* 3.1 (2010), pp. 285–296. URL: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R25.pdf.

[Bur+94] M. Burrows and D.J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Tech. rep. 124. Palo Alto, California 94301: Digital Systems Research Center, 10th May 1994.

[Cal+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 'Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency'. In: *SOSP*. Ed. by Ted Wobber and Peter Druschel. ACM, 2011, pp. 143–157. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043571.

[Cán+13] Rodrigo Cánovas and Alistair Moffat. 'Practical Compression for Multi-Alignment Genomic Files'. In: *Computer Science 2013 (ACSC 2013)*. Ed. by B. Thomas. Vol. 135. CRPIT. Adelaide,

Australia: ACS, 2013, pp. 51–60. URL: https://crpit.com/confpapers/CRPITV135Canovas.pdf.

[Cap59] Jack Capon. 'A Probabilistic Model for Run-Length Coding of Pictures'. In: *Information Theory, IRE Transactions on* 5.4 (Dec. 1959), pp. 157–163. ISSN: 0096-1000. DOI: 10.1109/TIT.1959.1057512.

[Car+91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. 'The information visualizer, an information workspace'. In: *CHI*. Ed. by Scott P. Robertson, Gary M. Olson, and Judith S. Olson. ACM, 1991, pp. 181–186. ISBN: 978-0-89791-383-6. DOI: 10.1145/108844.108874.

[CAS11] *CASAVA v1.8 User Guide*. Illumina, Inc. 2011. URL: http://biowulf.nih.gov/apps/CASAVA_UG_15011196B.pdf.

[CDH] *CDH*. Cloudera, Inc. URL: https://www.cloudera.com/content/cloudera/en/products/cdh.html.

[Cha+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 'Bigtable: A Distributed Storage System for Structured Data'. In: *OSDI*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 205–218. ISBN: 978-1-931971-47-8. URL: http://www.usenix.org/events/osdi06/tech/chang.html.

[Cha+11] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. 'Tenzing. A SQL Implementation On The MapReduce Framework'. In: *PVLDB* 4.12 (2011), pp. 1318–1327. URL: http://www.vldb.org/pvldb/vol4/p1318-chattopadhyay.pdf.

[Cha+12] Yu-Jung Chang, Chien-Chih Chen, Jan-Ming Ho, and Chuen-Liang Chen. '*De Novo* Assembly of High-Throughput Sequencing Data with Cloud Computing and New Operations on String Graphs'. In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. 2012, pp. 155–161. DOI: 10.1109/CLOUD.2012.123.

[Cha+74] Donald D. Chamberlin and Raymond F. Boyce. 'SEQUEL: A Structured English Query Language'. In: *SIGMOD Workshop, Vol. 1*. Ed. by Randall Rustin. ACM, 1974, pp. 249–264. DOI: 10.1145/800296.811515.

[Cha+98] Chee Yong Chan and Yannis E. Ioannidis. 'Bitmap Index Design and Evaluation'. In: *SIGMOD Conference*. Ed. by Laura M. Haas and Ashutosh Tiwary. ACM Press, 1998, pp. 355–366. ISBN: 978-0-89791-995-1. DOI: 10.1145/276304.276336.

[Che+12] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. 'Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads'. In: *PVLDB* 5.12 (2012), pp. 1802–1813. URL: http://vldb.org/pvldb/vol5/p1802_yanpeichen_vldb2012.pdf.

[Clo] *Cloudera, Inc.* URL: http://www.cloudera.com.

[Coc+10] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. 'The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants'. In: *Nucleic Acids Research* 38.6 (2010), pp. 1767–1771. DOI: 10.1093/nar/gkp1137.

[Coc11] Peter Cock. 'BGZF - Blocked, Bigger & Better GZIP!' In: *Blasted Bioinformatics!?* (8th Nov. 2011). URL: http://blastedbio.blogspot.com/2011/11/bgzf-blocked-bigger-better-gzip.html.

[Cod70] E. F. Codd. 'A Relational Model of Data for Large Shared Data Banks'. In: *Communications of the ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[Col05] Lasse Collin. *A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA*. 31st May 2005. URL: http://tukaani.org/lzma/benchmarks.html.

[Coo+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 'Benchmarking Cloud Serving Systems with YCSB'. In: *SoCC*. Ed. by Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum. ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152.

[Cop+85] George P. Copeland and Setrag Khoshafian. 'A Decomposition Storage Model'. In: *SIGMOD Conference*. Ed. by Shamkant B. Navathe. ACM Press, 1985, pp. 268–279. DOI: 10.1145/318898.318923.

[Dan+11] Petr Danecek, Adam Auton, Gonçalo R. Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, and Richard Durbin. 'The variant call format and VCFtools'. In: *Bioinformatics* 27.15 (2011), pp. 2156–2158. DOI: 10.1093/bioinformatics/btr330.

[Dar+03] Aaron E. Darling, Lucas Carey, and Wu-chun Feng. 'The Design, Implementation, and Evaluation of mpiBLAST'. In: ClusterWorld. 2003. URL: http://www.mpiblast.org/downloads/pubs/cwce03.pdf.

[dAra+11] Emerson de Araujo Macedo, Alba Cristina Magalhaes Alves de Melo, Gerson Henrique Pfitscher, and Azzedine Boukerche. 'Hybrid MPI/OpenMP Strategy for Biological Multiple Sequence Alignment with DIALIGN-TX in Heterogeneous Multicore Clusters'. In: *IPDPS Workshops*. IEEE, 2011, pp. 418–425. ISBN: 978-1-61284-425-1. DOI: 10.1109/IPDPS.2011.169.

[Dat06] C. J. Date. *Date on Database: Writings 2000–2006*. Apress, 2006. ISBN: 978-1-59059-746-0.

[Daw+09] Michael Dawson, Graeme Johnson, and Andrew Low. 'Best practices for using the Java Native Interface. Techniques and tools for averting the 10 most common JNI programming mistakes'. In: *IBM developerWorks* (7th July 2009). URL: https://www.ibm.com/developerworks/java/library/j-jni/.

[DCp] *DistCp Guide*. Version 1.2.1. URL: https://hadoop.apache.org/docs/r1.2.1/distcp.html.

[DDN] *DDN | SFA10K-X™*. DataDirect Networks, Inc. URL: https://www.ddn.com/products/sfa10000.

[DDR08] *Double Data Rate (DDR) SDRAM*. Tech. rep. JESD79F. JEDEC Solid State Technology Association, Feb. 2008. URL: http://www.jedec.org/standards-documents/docs/jesd-79f.

[DDR12] *DDR3 SDRAM Standard*. Tech. rep. JESD79-3F. JEDEC Solid State Technology Association, July 2012. URL: http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[Dea+04] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *OSDI*. USENIX Association, 2004, pp. 137–150. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

[Dea+13] Jeffrey Dean and Luiz André Barroso. 'The Tail at Scale'. In: *Communications of the ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794.

[Dea09] Jeffrey Dean. 'Designs, Lessons and Advice from Building Large Distributed Systems'. 2009. URL: https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[Den+07] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *Solid-State Circuits Society Newsletter, IEEE* 12.1 (2007), pp. 38–50. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785543.

[Deu+96] L. Peter Deutsch and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950 (Informational). Network Working Group, May 1996. URL: https://tools.ietf.org/html/rfc1950.

[Deu96a] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951 (Informational). Network Working Group, May 1996. URL: https://tools.ietf.org/html/rfc1951.

[Deu96b] L. Peter Deutsch. *GZIP file format specification version 4.3*. RFC 1952 (Informational). Network Working Group, May 1996. URL: https://tools.ietf.org/html/rfc1952.

[DeW+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 'Implementation Techniques for Main Memory Database Systems'. In: *SIGMOD Conference*. Ed. by Beatrice Yormark. ACM Press, 1984, pp. 1–8. DOI: 10.1145/602259.602261.

[DOI] *DOI® Handbook*. DOI: 10.1000/182.

[Dri] *Apache Drill*. The Apache Software Foundation. URL: https://incubator.apache.org/drill/.

[Duc03] David Duce, ed. *Portable Network Graphics (PNG) Specification*. 2nd ed. 10th Nov. 2003. URL: http://www.w3.org/TR/2003/REC-PNG-20031110/.

[Elm+10] Ahmed K. Elmagarmid and Divyakant Agrawal, eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010. ISBN: 978-1-4503-0032-2.

[Ens13] *Ensembl Glossary*. Version 73. Sept. 2013. URL: http://www.ensembl.org/info/website/glossary.html.

[Esm+11] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 'Dark silicon and the end of multicore scaling'. In: *ISCA*. Ed. by Ravi Iyer, Qing Yang, and Antonio González. ACM, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108.

[Fac] *Facebook*. URL: https://www.facebook.com.

[FAQ] *Frequently Asked Questions: Data File Formats*. UCSC Genome Bioinformatics. URL: http://www.genome.ucsc.edu/FAQ/FAQformat.html.

[Flo+11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 'Column-Oriented Storage Techniques for MapReduce'. In: *PVLDB* 4.7 (Apr. 2011), pp. 419–429. URL: http://www.vldb.org/pvldb/vol4/p419-floratou.pdf.

[Fri+76] Daniel P. Friedman and David S. Wise. 'CONS Should Not Evaluate its Arguments'. In: *ICALP*. 1976, pp. 257–284.

[Gai+13] Jean-loup Gailly and Mark Adler. *zlib Home Site*. Version 1.2.8. 28th Apr. 2013. URL: http://zlib.net.

[GAQ] Geraldine Van der Auwera. *Overview of Queue*. Broad Institute. URL: https://www.broadinstitute.org/gatk/guide/article?id=1306.

[Gat+09] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 'Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience'. In: *PVLDB* 2.2 (2009), pp. 1414–1425. URL: http://www.vldb.org/pvldb/2/vldb09-1074.pdf.

[Geo11] Lars George. *HBase: The Definitive Guide. Random Access to Your Planet-Size Data*. O'Reilly, 2011. ISBN: 978-1-449-39610-7.

[Ghe+03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 'The Google file system'. In: *SOSP*. Ed. by Michael L. Scott and Larry L. Peterson. ACM, 2003, pp. 29–43. ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945450.

[Gia99] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999. ISBN: 978-1-55860-497-1. URL: http://www.nobius.org/~dbg/practical-file-system-design.pdf.

[Gir] *Apache Giraph*. The Apache Software Foundation. URL: https://giraph.apache.org.

[GNU] *The GNU Operating System*. Free Software Foundation, Inc. URL: https://gnu.org.

[Goo] *Google*. URL: https://www.google.com.

[Gos+13] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, Feb. 2013. ISBN: 978-0-1332-6022-9.

[Gro+09] James R. Groff, Paul N. Weinberg, and Andrew J. Oppel. *SQL: The Complete Reference*. Third Edition. McGraw-Hill Osborne Media, Aug. 2009. ISBN: 978-0-0715-9255-0.

[Gut84] Antonin Guttman. 'R-Trees: A Dynamic Index Structure for Spatial Searching'. In: *SIGMOD Conference*. Ed. by Beatrice Yormark. ACM Press, 1984, pp. 47–57. DOI: 10.1145/602259.602266.

[Had] *Apache Hadoop*. The Apache Software Foundation. URL: https://hadoop.apache.org.

[HaL] *haloop - An modified version of Hadoop to support efficient iterative data processing on large commodity clusters*. URL: https://code.google.com/p/haloop/.

[Han+03] Richard A. Hankins and Jignesh M. Patel. 'Data Morphing: An Adaptive, Cache-Conscious Storage Technique'. In: *VLDB*. 2003, pp. 417–428. URL: http://www.vldb.org/conf/2003/papers/S13P03.pdf.

[Har+11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 'Toward Dark Silicon in Servers'. In: *IEEE Micro* 31.4 (2011), pp. 6–15. DOI: 10.1109/MM.2011.77.

[HAW13] *Pivotal HD: HAWQ. A true SQL engine for Hadoop*. White paper. Pivotal, Inc., 24th Apr. 2013. URL: http://gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf.

[HBa] *Apache HBase*. The Apache Software Foundation. URL: https://hbase.apache.org.

[HBA12] *[HBASE-5954] Allow proper fsync support for HBase*. Apache Software Foundation. 2012. URL: https://issues.apache.org/jira/browse/HBASE-5954.

[HBM] *Hadoop-BAM*. URL: http://sourceforge.net/projects/hadoop-bam/.

[HBP] *Hbase/PoweredBy*. URL: https://wiki.apache.org/hadoop/Hbase/PoweredBy.

[HBR] *The Apache HBase Reference Guide*. The Apache Software Foundation. URL: https://hbase.apache.org/book.html.

[He+11] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 'RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems'. In: *ICDE*. Ed. by Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan. IEEE Computer Society, 2011, pp. 1199–1208. ISBN: 978-1-4244-8958-9. DOI: 10.1109/ICDE.2011.5767933.

[Hee+12] Jeffrey Heer and Sean Kandel. 'Interactive analysis of big data'. In: *ACM Crossroads* 19.1 (2012), pp. 50–54. DOI: 10.1145/2331042.2331058.

[Hen+76] Peter Henderson and James H. Morris Jr. 'A Lazy Evaluator'. In: *POPL*. Ed. by Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman. ACM Press, 1976, pp. 95–103. DOI: 10.1145/800168.811543.

[HGS13] *HGST Ships 6TB Ultrastar® He6 Helium-filled Drives for High-density, Massive Scale-out Data Center Environments*. HGST, Inc. 4th Nov. 2013. URL: http://www.hgst.com/press-room/press-releases/hgst-ships-6TB-Ultrastar-HE6-helium-filled.

[HiJ] *LanguageManual Joins*. Apache Software Foundation. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins.

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 'Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center'. In: *NSDI*. Boston, MA, USA: USENIX Association, Apr. 2011, pp. 295–308. ISBN: 978-931971-84-3. URL: https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Hindman_new.pdf.

[Hiv] *Apache Hive*. The Apache Software Foundation. URL: https://hive.apache.org.

[HIV10] *[HIVE-1402] Add parallel ORDER BY to Hive*. Apache Software Foundation. 2010. URL: https://issues.apache.org/jira/browse/HIVE-1402.

[HNL13] *Native Libraries Guide*. Version 2.2.0. 7th Oct. 2013. URL: https://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-common/NativeLibraries.html.

[Hof13] Lars Hofhansl. 'Protecting HBase against data center outages'. In: *HBase* (2nd July 2013). URL: http://hadoop-hbase.blogspot.com/2013/07/protected-hbase-against-data-center.html.

[Hor] *Hortonworks Inc.* URL: http://www.hortonworks.com.

[IEE08] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[ImB13] *[#IMPALA-482] "block size is too big" error with Snappy-compressed RCFile containing null*. 2013. URL: https://issues.cloudera.org/browse/IMPALA-482.

[ImC] *Post-Installation Configuration for Impala*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.0.1/Installing-and-Using-Impala/ciiu_config_performance.html.

[ImD] *Cloudera Impala Documentation*. Cloudera, Inc. URL: https://www.cloudera.com/content/support/en/documentation/cloudera-impala/cloudera-impala-documentation-v1-latest.html.

[ImF] *New Features in Impala*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Cloudera-Impala-Release-Notes/cirn_new_features.html.

[ImG] *Cloudera Impala*. URL: https://github.com/cloudera/impala.

[ImL] *Impala SQL Language Elements*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.0.1/Installing-and-Using-Impala/ciiu_langref_sql.html.

[Imp] *Introducing Impala*. Cloudera, Inc. URL: http://cloudera.com/impala/.

[IMP13] *IMPALA-482: "block size is too big" error with Snappy-compressed RCFi....* 2013. URL: https://github.com/cloudera/impala/commit/7730fae3365eeeaaa5d0d991d68c775c4ed6e86a.

[ImQ] *Cloudera Impala Frequently Asked Questions*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.0.1/Cloudera-Impala-Frequently-Asked-Questions/Cloudera-Impala-Frequently-Asked-Questions.html.

[ImT13] *Cloudera Impala: How exactly does a multi-level execution tree improve Impala Query performance?* 2013. URL: https://www.quora.com/Cloudera-Impala/How-exactly-does-a-multi-level-execution-tree-improve-Impala-Query-performance?share=1.

[ImU] *Unsupported Language Elements*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.0.1/Installing-and-Using-Impala/ciiu_langref_unsupported.html.

[Int05] *Excerpts from A Conversation with Gordon Moore: Moore's Law*. Intel Corporation, 2005. URL: https://web.archive.org/web/20130420194212/http://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.

[IRC] *Using RCFile, SequenceFile, or Text Files*. URL: https://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.0.1/Installing-and-Using-Impala/ciiu_rcfile.html.

[ISO92] International Organization for Standardization. *ISO/IEC 9075:1992. Information technology – Database languages – SQL*. Geneva, Switzerland, 1992. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=16663.

[Jun+11] F.P. Junqueira, B.C. Reed, and M. Serafini. 'Zab: High-performance broadcast for primary-backup systems'. In: *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. 2011, pp. 245–256. DOI: 10.1109/DSN.2011.5958223.

[Ken+02] W. James Kent, Charles W. Sugnet, Terrence S. Furey, Krishna M. Roskin, Tom H. Pringle, Alan M. Zahler, and David Haussler. 'The Human Genome Browser at UCSC'. In: *Genome Research* 12.6 (2002), pp. 996–1006. DOI: 10.1101/gr.229102.

[Kes13] Justin Kestelyn. 'Introducing Parquet: Efficient Columnar Storage for Apache Hadoop'. In: *Cloudera Blog* (13th Mar. 2013). URL: https://blog.cloudera.com/blog/2013/03/introducing-parquet-columnar-storage-for-apache-hadoop/.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 978-0-201-03803-3.

[Kor+12] Marcel Kornacker and Justin Erickson. 'Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real'. In: *Cloudera Blog* (24th Oct. 2012). URL: https://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/.

[Kre+11] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. 'Disks are like snowflakes: no two are alike'. In: *Proceedings of the 13th USENIX conference on Hot topics in operating systems*. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=1991596.1991615.

[Kry+09] M.H. Kryder and Chang Soo Kim. 'After Hard Drives — What Comes Next?' In: *Magnetics, IEEE Transactions on* 45.10 (2009), pp. 3406–3413. ISSN: 0018-9464. DOI: 10.1109/TMAG.2009.2024163.

[Lam98] Leslie Lamport. 'The Part-Time Parliament'. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229.

[Lan+09] Ben Langmead, Michael Schatz, Jimmy Lin, Mihai Pop, and Steven Salzberg. 'Searching for SNPs with cloud computing'. In: *Genome Biology* 10.11 (2009), R134. ISSN: 1465-6906. DOI: 10.1186/gb-2009-10-11-r134. PMID: 19930550.

[Lan+10] Ben Langmead, Kasper Hansen, and Jeffrey Leek. 'Cloud-scale RNA-sequencing differential expression analysis with Myrna'. In: *Genome Biology* 11.8 (2010), R83. ISSN: 1465-6906. DOI: 10.1186/gb-2010-11-8-r83. PMID: 20701754.

[Lat+04] Chris Lattner and Vikram S. Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *CGO*. IEEE Computer Society, 2004, pp. 75–88. ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665.

[Leb13] Scott Leberknight. "Cloudera's Impala". 9th July 2013. URL: http://www.slideshare.net/cloudera/impala-v1update130709222849phpapp01.

[Li+09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. 'The Sequence Alignment/Map format and SAMtools'. In: *Bioinformatics* 25.16 (2009), pp. 2078–2079. DOI: 10.1093/bioinformatics/btp352.

[Li13] Nong Li. 'Inside Cloudera Impala: Runtime Code Generation'. In: *Cloudera Blog* (11th Feb. 2013). URL: https://blog.cloudera.com/blog/2013/02/inside-cloudera-impala-runtime-code-generation/.

[LiI] *LinkedIn*. URL: https://www.linkedin.com.

[Lin+11] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. 'Full-text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics'. In: *Proceedings of the second international workshop on MapReduce and its applications*. MapReduce

'11. San Jose, California, USA: ACM, 2011, pp. 59–66. ISBN: 978-1-4503-0700-0. DOI: 10.1145/1996092.1996105.

[LLV] *The LLVM Compiler Infrastructure Project*. URL: http://llvm.org.

[Lov28] H. P. Lovecraft. 'The Call of Cthulhu'. In: *Weird Tales* 11.2 (Feb. 1928). Ed. by Farnsworth Wright. URL: https://en.wikisource.org/wiki/The_Call_of_Cthulhu.

[Lus] *Lustre*. Xyratex. URL: http://lustre.org.

[Mah] Matt Mahoney. *ZPAQ*. URL: http://mattmahoney.net/dc/zpaq.html.

[Mal+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 'Pregel: A System for Large-Scale Graph Processing'. In: *SIGMOD Conference*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184.

[MaR] *MapR Technologies, Inc.* URL: http://www.mapr.com.

[Mar13] Vivien Marx. 'Biology: The big challenges of big data'. In: *Nature* 498.7453 (13th June 2013). Technology Feature, pp. 255–260. ISSN: 0028-0836. DOI: 10.1038/498255a.

[Mar96] Robert C. Martin. *Re: Challenge: Modelling Human Understanding (Was: Re: Programmers to dumb...?)* On Usenet, newsgroup comp.object. Message-ID: rmartin-1108962342100001@vh1-006.wwa.com. 11th Aug. 1996.

[Mas] Matt Massie. *ADAM: Datastore Alignment Map*. URL: https://github.com/massie/adam/.

[Mat+08] Andréa M. Matsunaga, Maurício O. Tsugawa, and José A. B. Fortes. 'CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications'. In: *eScience*. IEEE Computer Society, 2008, pp. 222–229. DOI: 10.1109/eScience.2008.62.

[McK+09] Marshall Kirk McKusick and Sean Quinlan. 'GFS: Evolution on Fast-forward'. In: *Queue* 7.7 (Aug. 2009), 10:10–10:20. ISSN: 1542-7730. DOI: 10.1145/1594204.1594206.

[McK+10] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. 'The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data'. In: *Genome Research* 20.9 (2010), pp. 1297–1303. DOI: 10.1101/gr.107524.110.

[Mel+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 'Dremel: Interactive Analysis of Web-Scale Datasets'. In: *PVLDB* 3.1 (2010), pp. 330–339. URL: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R29.pdf.

[Mes] *Apache Mesos*. The Apache Software Foundation. URL: https://mesos.apache.org.

[Moh+13] Nabeel M. Mohamed, Heshan Lin, and Wu-chun Feng. 'Accelerating Data-Intensive Genome Analysis in the Cloud'. In: *5th International Conference on Bioinformatics and Computational Biology (BICoB)*. Honolulu, Hawaii, USA, Mar. 2013. URL: http://synergy.cs.vt.edu/pubs/papers/nabeel-bicob13-genome-analysis-cloud.pdf.

[Mon+13] Alberto Montañola, Concepcio Roig, and Porfidio Hernández. 'Pairwise Sequence Alignment Method for Distributed Shared Memory Systems'. In: *PDP*. IEEE Computer Society, 2013, pp. 432–436. ISBN: 978-1-4673-5321-2. DOI: 10.1109/PDP.2013.69.

[Moo65] Gordon E. Moore. 'Cramming more components onto integrated circuits'. In: *Electronics* 38.8 (Apr. 1965).

[Moo75] G.E. Moore. 'Progress in digital integrated electronics'. In: *Electron Devices Meeting, 1975 International*. Vol. 21. 1975, pp. 11–13.

[MPI93] 'MPI: A Message Passing Interface'. In: *SC*. Ed. by Bob Borchers and Dona Crawford. The MPI Forum. IEEE Computer Society / ACM, 1993, pp. 878–883. ISBN: 978-0-8186-4340-8. DOI: 10.1145/169627.169855.

[Mur+13] Arun C. Murthy and Bikas Saha. 'Apache Tez: Accelerating Hadoop Query Processing'. 9th July 2013. URL: http://www.slideshare.net/Hadoop_Summit/murhty-saha-june26255pmroom212.

[Mur12] Arun Murthy. 'Apache Hadoop YARN – Background and an Overview'. In: (7th Aug. 2012). URL: https://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/.

[Ngu+11] Tung Nguyen, Weisong Shi, and Douglas Ruden. 'CloudAligner: A fast and full-featured MapReduce based tool for sequence mapping'. In: *BMC Research Notes* 4.1 (2011), p. 171. ISSN: 1756-0500. DOI: 10.1186/1756-0500-4-171. PMID: 21645377.

[Nie+12] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. 'Hadoop-BAM: directly manipulating next generation sequencing data in the cloud'. In: *Bioinformatics* 28.6 (2012), pp. 876–877. DOI: 10.1093/bioinformatics/bts054.

[Nie11] Matti Niemenmaa. 'Interactivity for Big Data: Preprocessing genomic data with MapReduce'. Bachelor's Thesis. Aalto University School of Science, 4th May 2011. URL: https://deewiant.iki.fi/writings/thesis-bachelor.pdf.

[Nit+91] Bill Nitzberg and Virginia Mary Lo. 'Distributed Shared Memory: A Survey of Issues and Algorithms'. In: *IEEE Computer* 24.8 (1991), pp. 52–60. DOI: 10.1109/2.84877.

[Nor+13] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. 'BioPig: A Hadoop-based Analytic Toolkit for Large-Scale Sequence Data'. In: *Bioinformatics* (2013). DOI: 10.1093/bioinformatics/btt528.

[Nov13] Jordan Novet. 'Facebook unveils Presto engine for querying 250 PB data warehouse'. In: *GigaOM* (6th June 2013). URL: http://gigaom.com/2013/06/06/facebook-unveils-presto-engine-for-querying-250-pb-data-warehouse/.

[Nut] *Apache Nutch*. The Apache Software Foundation. URL: https://nutch.apache.org.

[Obe] Markus F.X.J. Oberhumer. *LZO real-time data compression library*. URL: http://www.oberhumer.com/opensource/lzo/.

[OCo+10] Brian D. O'Connor, Barry Merriman, and Stanley F. Nelson. 'SeqWare Query Engine: storing and searching sequence data in the cloud'. In: *BMC Bioinformatics* 11.S-12 (2010), S2. DOI: 10.1186/1471-2105-11-S12-S2.

[Ode+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. *An Overview of the Scala Programming Language*. Tech. rep. LAMP-REPORT-2006-001. 1015 Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, 2006. URL: http://scala-lang.org/docu/files/ScalaOverview.pdf.

[ODe13] Kevin O'Dell. *How-to: Select the Right Hardware for Your New Hadoop Cluster*. 28th Aug. 2013. URL: http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/.

[Ols+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 'Pig Latin: A Not-So-Foreign Language for Data Processing'. In: *SIGMOD Conference*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 1099–1110. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376726.

[OMa13] Owen O'Malley. 'Optimizing Hive Queries'. 27th Mar. 2013. URL: http://www.slideshare.net/oom65/optimize-hivequeriespptx.

[ORC13] *[HIVE-3874] Create a new Optimized Row Columnar file format for Hive*. Apache Software Foundation. 2013. URL: https://issues.apache.org/jira/browse/HIVE-3874.

[ORM] *LanguageManual ORC*. Apache Software Foundation. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC.

[Ous+11] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 'The case for RAMCloud'. In: *Communications of*

the ACM 54.7 (2011), pp. 121–130. DOI: 10.1145/1965724.1965751.

[Pab+13] Stephan Pabinger, Andreas Dander, Maria Fischer, Rene Snajder, Michael Sperk, Mirjana Efremova, Birgit Krabichler, Michael R. Speicher, Johannes Zschocke, and Zlatko Trajanoski. 'A survey of tools for variant analysis of next-generation genome sequencing data'. In: *Briefings in Bioinformatics* (2013). DOI: 10.1093/bib/bbs086.

[Par] *Parquet: Columnar Storage for Hadoop*. URL: http://parquet.io.

[Pat04] David A. Patterson. 'Latency lags bandwith'. In: *Communications of the ACM* 47.10 (2004), pp. 71–75. DOI: 10.1145/1022594.1022596.

[Pav+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 'A Comparison of Approaches to Large-Scale Data Analysis'. In: *SIGMOD Conference*. Ed. by Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul. ACM, 2009, pp. 165–178. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559865.

[Pav13] Igor Pavlov. *LZMA SDK (Software Development Kit)*. 2013. URL: http://www.7-zip.org/sdk.html.

[Pea+88] William R. Pearson and David J. Lipman. 'Improved tools for biological sequence comparison'. In: *PNAS* 85.8 (Apr. 1988), pp. 2444–2448. PMID: 3162770.

[Pet+61] W. W. Peterson and D. T. Brown. 'Cyclic Codes for Error Detection'. In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1961.287814.

[Pig] *Apache Pig*. The Apache Software Foundation. URL: https://pig.apache.org.

[Pik+93] Rob Pike and Ken Thompson. 'Hello World or Καλημέρα κόσμε or こんにちは 世界'. In: *USENIX Winter*. 1993, pp. 43–50. URL: http://doc.cat-v.org/plan_9/4th_edition/papers/utf.

[Pir+11a] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. 'MapReducing a Genomic Sequencing Workflow'. In: *Proceedings of the second international workshop on MapReduce and its applications*. MapReduce '11. San Jose, California, USA: ACM, 2011, pp. 67–74. ISBN: 978-1-4503-0700-0. DOI: 10.1145/1996092.1996106.

[Pir+11b] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. 'SEAL: a distributed short read mapping and duplicate removal tool'. In: *Bioinformatics* 27.15 (2011), pp. 2159–2160. DOI: 10.1093/bioinformatics/btr325.

[Pla+11] Hasso Plattner and Alexander Zeier. *In-Memory Data Management. An Inflection Point for Enterprise Applications*. Berlin, Heidelberg: Springer-Verlag, 2011. ISBN: 978-3-642-19362-0. DOI: 10.1007/978-3-642-19363-7.

[Pra+12] Brian Pratt, J. Jeffry Howbert, Natalie I. Tasman, and Erik J. Nilsson. 'MR-Tandem: parallel X!Tandem using Hadoop MapReduce on Amazon Web Services'. In: *Bioinformatics* 28.1 (2012), pp. 136–137. DOI: 10.1093/bioinformatics/btr615.

[Pre] *Presto | Distributed SQL Query Engine for Big Data*. Facebook. URL: http://prestodb.io.

[Pri71] Ludwig Prinn. *De Vermiis Mysteriis*. Ed. by Robert Bloch and H. P. Lovecraft. 1271.

[Pyt] *Python Programming Language*. Python Software Foundation. URL: http://www.python.org/.

[Qui+10] Aaron R. Quinlan and Ira M. Hall. 'BEDTools: a flexible suite of utilities for comparing genomic features'. In: *Bioinformatics* 26.6 (2010), pp. 841–842. DOI: 10.1093/bioinformatics/btq033.

[Red] *Amazon Redshift*. Amazon Web Services, Inc. URL: https://aws.amazon.com/redshift/.

[Ree10] Benjamin Reed. "Hadoop @ Yahoo! an admin's perspective". 2010. URL: http://www.cs.duke.edu/smdb10/_files/toc_data/SMDB/panel/reed.pdf.

[Rey98] Carson Reynolds. 'As We May Communicate'. In: *ACM SIGCHI Bulletin* 30.3 (July 1998), pp. 40–44. ISSN: 0736-6906. DOI: 10.1145/565711.565714.

[Rez+06] S. Rezaei, M. Monwar, and J. Bai. 'Performance Comparison of MPI-Based Parallel Multiple Sequence Alignment Algorithm Using Single and Multiple Guide Trees'. In: *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on.* Vol. 1. 2006, pp. 595–600. DOI: 10.1109/COGINF.2006.365552.

[Rob+11] Thomas Robinson, Sarah Killcoyne, Ryan Bressler, and John Boyle. 'SAMQA: error classification and validation of high-throughput sequenced read data'. In: *BMC Genomics* 12.1 (2011), p. 419. ISSN: 1471-2164. DOI: 10.1186/1471-2164-12-419. PMID: 21851633.

[Rya13] Dmitriy Ryaboy. 'Announcing Parquet 1.0: Columnar Storage for Hadoop'. In: *Twitter Blog* (30th July 2013). URL: https://blog.twitter.com/2013/announcing-parquet-10-columnar-storage-for-hadoop.

[Sag90] Carl Sagan. 'Why We Need To Understand Science'. In: *The Skeptical Inquirer* 14 (3 1990). URL: http://www.csicop.org/si/show/why_we_need_to_understand_science.

[SAM13] *Sequence Alignment/Map Format Specification.* Tech. rep. Version 321f786. The SAM/BAM Format Specification Working Group, 29th May 2013. URL: http://samtools.sourceforge.net/SAMv1.pdf.

[Sch+] Michael Schatz, Jeremy Chambers, Avijit Gupta, Rushil Gupta, David Kelley, Jeremy Lewi, Deepak Nettem, Dan Sommer, and Mihai Pop. *Contrail: Assembly of Large Genomes using Cloud Computing.* URL: http://sourceforge.net/apps/mediawiki/contrail-bio/.

[Sch+12] Sebastian Schönherr, Lukas Forer, Hansi Weißensteiner, Florian Kronenberg, Günther Specht, and Anita Kloss-Brandstätter. 'Cloudgene: A graphical execution platform for MapReduce programs on private and public clouds'. In: *BMC Bioinformatics* 13 (2012), p. 200. DOI: 10.1186/1471-2105-13-200.

[Sch+13a] André Schumacher, Luca Pireddu, Aleksi Kallio, Matti Niemenmaa, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. 'Scripting for large-scale sequencing based on Hadoop'. In: *EMBnet.journal* 19.A (2013). URL: http://journal.embnet.org/index.php/embnetjournal/article/view/628.

[Sch+13b] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. 'SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop'. In: *Bioinformatics* (2013). DOI: 10.1093/bioinformatics/btt601.

[Sch03] Philip Schwan. 'Lustre: Building a File Fystem for 1,000-node Clusters'. In: *Proceedings of the 2003 Linux Symposium*. Cluster File Systems, Inc. 2003. URL: https://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf.

[Sch09] Michael C. Schatz. 'CloudBurst: highly sensitive read mapping with MapReduce'. In: *Bioinformatics* 25.11 (2009), pp. 1363–1369. DOI: 10.1093/bioinformatics/btp236.

[SDR94] *Synchronous Dynamic Random Access Memory (SDRAM)*. June 1994. URL: http://www.jedec.org/standards-documents/docs/sdram-311.

[SeF] *SequenceFile*. API documentation. Version 2.2.0. URL: https://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/io/SequenceFile.html.

[Seq] *SeqPig Manual*. URL: http://seqpig.sourceforge.net.

[Sew] Julian Seward. *bzip2*. URL: http://www.bzip.org.

[Sha] *Shark*. URL: https://github.com/amplab/shark/wiki.

[Sha+12] Weiyi Shang, Bram Adams, and Ahmed E. Hassan. 'Using Pig as a data preparation language for large-scale mining software repositories studies: An experience report'. In: *Journal of Systems and Software* 85.10 (2012), pp. 2195–2204. DOI: 10.1016/j.jss.2011.07.034.

[SHA12] *[SHARK-96] Port partial dag execution back to trunk*. 2012. URL: https://spark-project.atlassian.net/browse/SHARK-96.

[SHA13] *[SHARK-147] map join runs abnormal in cluster mode*. 2013. URL: https://spark-project.atlassian.net/browse/SHARK-147.

[Sha48] C. E. Shannon. 'A Mathematical Theory of Communication'. In: *Bell System Technical Journal* 27 (July and October 1948), pp. 379–423, 623–656. URL: http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html.

[ShC] *Compatibility with Apache Hive*. URL: https://github.com/amplab/shark/wiki/Compatibility-with-Apache-Hive.

[Sim71] Herbert A. Simon. 'Designing Organizations for an Information-Rich World'. In: *Computers, Communication, and the Public Interest*. Ed. by Martin Greenberger. The Johns Hopkins Press, 1971, pp. 40–41. ISBN: 978-0-8018-1135-7.

[Sna] *snappy - A fast compressor/decompressor*. URL: https://code.google.com/p/snappy/.

[Spa] *Spark*. UC Berkeley AMPLab. URL: http://www.spark-project.org.

[SPD] URL: https://github.com/rxin/shark/tree/partial_dag.

[Ste10] Lincoln Stein. 'The case for cloud computing in genome informatics'. In: *Genome Biology* 11.5 (2010), p. 207. ISSN: 1465-6906. DOI: 10.1186/gb-2010-11-5-207. PMID: 20441614.

[StI] *SQL-in-Hadoop : The Stinger Initiative*. Hortonworks Inc. URL: https://hortonworks.com/stinger/.

[Sto+10] Inc. Storspeed and NetApp. *NFS Version 4 Minor Version 1*. Ed. by S. Shepler, M. Eisler, and D. Noveck. RFC 5661 (Proposed Standard). Internet Engineering Task Force (IETF), Jan. 2010. URL: https://tools.ietf.org/html/rfc5661.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Fourth edition. Addison-Wesley Professional, May 2013. ISBN: 978-0-3215-6384-2.

[Sun90] Vaidy S. Sunderam. 'PVM: A Framework for Parallel Distributed Computing'. In: *Concurrency - Practice and Experience* 2.4 (1990), pp. 315–339. DOI: 10.1002/cpe.4330020404.

[Sut09] Herb Sutter. 'The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software'. In: (Aug. 2009–). URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[Tay10] Ronald C. Taylor. 'An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics'. In: *BMC Bioinformatics* 11.S-12 (2010), S1. DOI: 10.1186/1471-2105-11-S12-S1.

[TeH] *Apache Tez*. Hortonworks Inc. URL: https://hortonworks.com/hadoop/tez/.

[Tez] *Tez*. The Apache Software Foundation. URL: http://tez.incubator.apache.org/.

[Thu+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 'Hive - A Warehousing Solution Over a Map-Reduce Framework'. In: *PVLDB* 2.2 (2009), pp. 1626–1629. URL: http://www.vldb.org/pvldb/2/vldb09-938.pdf.

[Thu+10a] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 'Hive – A Petabyte Scale Data Warehouse Using Hadoop'. In: *ICDE*. Ed. by Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras. IEEE, 2010, pp. 996–1005. ISBN: 978-1-4244-5444-0. DOI: 10.1109/ICDE.2010.5447738.

[Thu+10b] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 'Data Warehousing and Analytics Infrastructure at Facebook'. In: *SIGMOD Conference*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 1013–1020. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807278.

[TPC] *TPC-DS*. URL: http://www.tpc.org/tpcds/.

[Tra13] Martin Traverso. 'Presto: Interacting with petabytes of data at Facebook'. In: (6th Nov. 2013). URL: https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920.

[Tre13] *Trevni Specification*. Version 0.1. Apache Software Foundation. 21st Feb. 2013. URL: https://avro.apache.org/docs/1.7.4/trevni/spec.html.

[TrG] URL: https://github.com/cutting/trevni.

[TrH12] *[HIVE-3585] Integrate Trevni as another columnar oriented file format*. Apache Software Foundation. 2012. URL: https://issues.apache.org/jira/browse/HIVE-3585.

[Tuf90] Edward R. Tufte. *Envisioning Information*. Graphics Press, May 1990, p. 51. ISBN: 978-0-9613921-1-6.

[Twi] *Twitter*. URL: https://twitter.com.

[UTF] *UTF-8, UTF-16, UTF-32 & BOM*. Unicode, Inc. URL: http://www.unicode.org/faq/utf_bom.html.

[Wad71] C. P. Wadsworth. 'Semantics and Pragmatics of the Lambda-Calculus.' Doctoral thesis. University of Oxford, 1971.

[Wal05] Chip Walter. "Kryder's Law". In: *Scientific American* 293 (2 2005), pp. 32–33. DOI: 10.1038/scientificamerican0805-32.

[Wat12] John K. Waters. "Apache Hadoop Community Promotes YARN – But Don't Call it MapReduce 2". In: *WatersWorks* (15th Aug. 2012). URL: http://adtmag.com/blogs/watersworks/2012/08/apache-yarn-promotion.aspx.

[WAz] *Windows Azure*. Microsoft Corporation. URL: https://www.windowsazure.com.

[Wei10] Kevin Weil. 'Hadoop at Twitter'. 2010. URL: http://www.slideshare.net/kevinweil/hadoop-at-twitter-hadoop-summit-2010.

[Whi09] Tom White. *Hadoop: The Definitive Guide. MapReduce for the Cloud*. O'Reilly, 2009, pp. I–XIX, 1–501. ISBN: 978-0-596-52197-4.

[Wie] Dag Wieërs. *Dstat: Versatile resource statistics tool*. URL: http://dag.wieers.com/home-made/dstat/.

[Wie+75] Gio Wiederhold, James F. Fries, and Stephen Weyl. 'Structured organization of clinical data bases'. In: *AFIPS National Computer Conference*. Vol. 44. AFIPS Conference Proceedings. AFIPS Press, 1975, pp. 479–485. DOI: 10.1145/1499949.1500043.

[Wir95] Niklaus Wirth. 'A Plea for Lean Software'. In: *IEEE Computer* 28.2 (Feb. 1995), pp. 64–68. DOI: 10.1109/2.348001.

[Wri] *Writable*. API documentation. Version 2.2.0. URL: https://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/io/Writable.html.

[Xin+12] Reynold Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 'Shark: SQL and Rich Analytics at Scale'. In: *CoRR* abs/1211.6176 (2012). arXiv: 1211.6176 [cs.DB].

[Ya!] *Yahoo!* URL: http://www.yahoo.com.

[Yor84] Beatrice Yormark, ed. *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*. ACM Press, 1984.

[YRN] *Hadoop YARN. A next-generation framework for Hadoop data processing*. Hortonworks Inc. URL: https://hortonworks.com/hadoop/yarn/.

[YRN13] *Apache Hadoop NextGen MapReduce (YARN)*. Version 2.2.0. 7th Oct. 2013. URL: https://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 'Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing'. In: *NSDI*. San Jose, CA, USA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[Zoo] *Apache ZooKeeper*. The Apache Software Foundation. URL: https://zookeeper.apache.org.

[Zop] *zopfli - Zopfli Compression Algorithm*. URL: https://code.google.com/p/zopfli/.