Table of Contents

Cover 1.1

Part I. Hadoop Fundamentals 1.2

- Chapter 1. Meet Hadoop 1.2.1
  - Data! 1.2.1.1
  - Data Storage and Analysis 1.2.1.2
  - Querying All Your Data 1.2.1.3
  - Beyond Batch 1.2.1.4
  - Comparison with Other Systems 1.2.1.5
    - Relational Database Management Systems 1.2.1.5.1
    - Grid Computing 1.2.1.5.2
    - Volunteer Computing 1.2.1.5.3
  - A Brief History of Apache Hadoop 1.2.1.6
  - What's in This Book 1.2.1.7
- Chapter 2. MapReduce 1.2.2
  - A Weather Dataset 1.2.2.1
    - Data Format 1.2.2.1.1
  - Analyzing the Data with Unix Tools 1.2.2.2
  - Analyzing the Data with Hadoop 1.2.2.3
    - Map and Reduce 1.2.2.3.1
    - Java MapRedece 1.2.2.3.2
  - Scaling Out 1.2.2.4
    - Data Flow 1.2.2.4.1
    - Combiner Functions 1.2.2.4.2
    - Running a Distributed MapReduce Job 1.2.2.4.3
  - Hadoop Streaming 1.2.2.5
    - Ruby 1.2.2.5.1
    - Python 1.2.2.5.2
- Chapter 3. The Hadoop Distributed Filesystem 1.2.3

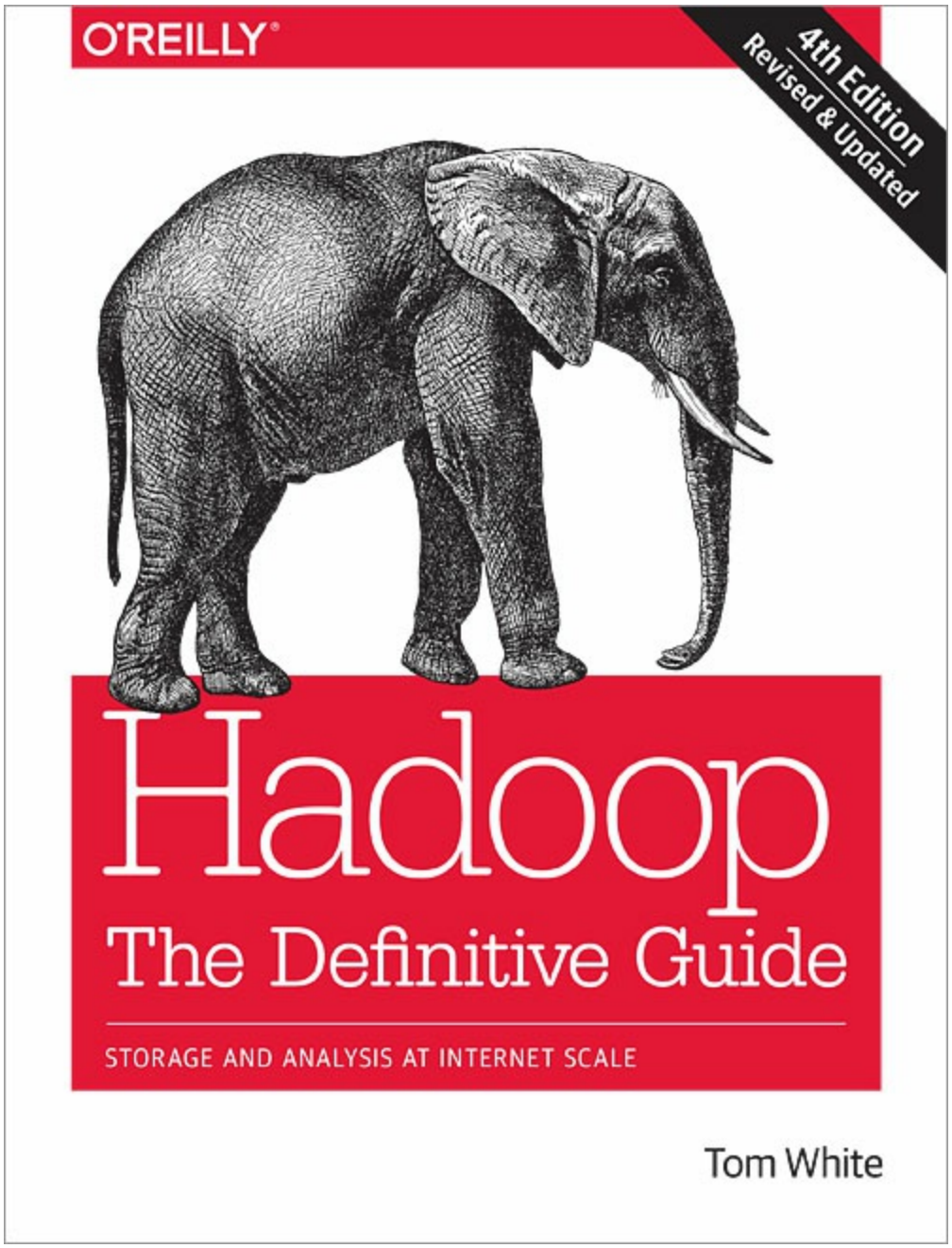
THe Design of HDFS 1.2.3.1
HDFS Concepts 1.2.3.2
Blocks 1.2.3.2.1
Namenodes and Datanodes 1.2.3.2.2
Block Caching 1.2.3.2.3
HDFS Federation 1.2.3.2.4
HDFS High Availability 1.2.3.2.5
The Command-Line Interface 1.2.3.3
Basic FileSystem Operations 1.2.3.3.1
Hadoop Filesystems 1.2.3.4
Interfaces 1.2.3.4.1
The Java Interface 1.2.3.5
Reading Data from a Hadoop URL 1.2.3.5.1
Reading Data Using the FileSystem API 1.2.3.5.2
Writing Data 1.2.3.5.3
Directories 1.2.3.5.4
Querying the Filesystem 1.2.3.5.5
Deleting Data 1.2.3.5.6
Data Flow 1.2.3.6
Anatomy of a File Read 1.2.3.6.1
Anatomy of a File Writw 1.2.3.6.2
Coherency Model 1.2.3.6.3
Parallel Copying with distcp 1.2.3.7
Keeping an HDFS Cluster Balanced 1.2.3.7.1
Chapter 4. YARN 1.2.4
Anatomy of a YARN Application Run 1.2.4.1
Resource Requests 1.2.4.1.1
Application Lifespan 1.2.4.1.2
Building YARN Application 1.2.4.1.3
Scheduling in YARN 1.2.4.2
Scheduler Options 1.2.4.2.1
Capacity Scheduler Configuration 1.2.4.2.2

Fair Schedular Configuration 1.2.4.2.3
Delay Scheduling 1.2.4.2.4
Dominant Resource Fairness 1.2.4.2.5
Futher Reading 1.2.4.3
Part II. MapReduce 1.3
Chapter 6. Developing a MapReduce Application 1.3.1
The Configuration API 1.3.1.1
Combining Resources 1.3.1.1.1
variable Expansion 1.3.1.1.2
Setting Up the Development Environment 1.3.1.2
Managing Configuration 1.3.1.2.1
GenericOptionsParser, Tool, and ToolRunner 1.3.1.2.2
Writing a Unit Test with MRUnit 1.3.1.3
Mapper 1.3.1.3.1
Reducer 1.3.1.3.2
Running Locally on Test Data 1.3.1.4
Running a Job in a Local Job Runner 1.3.1.4.1
Testing the Driver 1.3.1.4.2
Running on a Cluster 1.3.1.5
Packing a Job 1.3.1.5.1
Luanching a Job 1.3.1.5.2
The MapReduce Web UI 1.3.1.5.3
retrieving the Results 1.3.1.5.4
Debugging a Job 1.3.1.5.5
hadoop Logs 1.3.1.5.6
Remote Debugging 1.3.1.5.7
Tuning a Job 1.3.1.6
Profiling Tasks 1.3.1.6.1
MapReduce Workflows 1.3.1.7
Decomposing a Problem into MapReduce Jobs 1.3.1.7.1
JobControl 1.3.1.7.2
Apache Oozie 1.3.1.7.3

Part IV. Related Projects 1.4
Chapter 19. Spark 1.4.1
Installing Spark 1.4.1.1
An Example 1.4.1.2
Spark Application, Jobs, Stages, and Tasks 1.4.1.2.1
A Scala Standalong Application 1.4.1.2.2
A Java Example 1.4.1.2.3
A Python Example 1.4.1.2.4
Resilient Distributed Datasets 1.4.1.3
Creation 1.4.1.3.1
Transformations and Actions 1.4.1.3.2
Persistence 1.4.1.3.3
Serialization 1.4.1.3.4
Shared Variables 1.4.1.4
Broadcast Variables 1.4.1.4.1
Accumulators 1.4.1.4.2
Anatomy of a Spark Job Run 1.4.1.5
Job Submission 1.4.1.5.1
DAG Construction 1.4.1.5.2
Task Scheduling 1.4.1.5.3
Task Execution 1.4.1.5.4
Execution and Cluster Managers 1.4.1.6
Spark on YARN 1.4.1.6.1
Futher Reading 1.4.1.7

Hadoop: The Definitive Guide 4th Edition

Read on gh-pages | Read on GitBook



Part I. Hadoop Fundamentals

Chatper 1. Meet Hadoop

In pioneer days从前 they used oxen公牛 for heavy pulling, and when one ox couldn’t budge移动 a log原木, they didn’t try to grow a larger ox. We shouldn’t be trying for bigger computers, but for more systems of computers.

—Grace Hopper

Data!

We live in the data age. It's not easy to measure测量 the total volume of data stored electronically, but an IDC国际文献资料中心 estimate put the size of the "digital universe" at 4.4 zettabytes泽字节(ZB) in 2013 and is forecasting a tenfold growth by 2020 to 44 zettabytes.[1] A zettabyte is 1021 bytes, or equivalently相等地 one thousand exabytes艾字节(EB), one million petabytes拍字节(PB), or one billion terabytes太字节(TB). That's more than one disk drive for every person in the world.

This flood of data is coming from many sources. Consider the following:[2]

- The New York Stock Exchangen纽约股票交易所 generates about 4−5 terabytes of data per day.
- Facebook hosts more than 240 billion photos, growing at 7 petabytes per month.
- Ancestry.com, the genealogy site, stores around 10 petabytes of data.
- The Internet Archive互联网档案馆 stores around 18.5 petabytes of data.
- The Large Hadron Collider大型强子对撞机 near Geneva日内瓦, Switzerland瑞士, produces about 30 petabytes of data per year.

So there's a lot of data out there. But you are probably wondering how it affects you. Most of the data is locked up in the largest web properties (like search engines) or in scientific or financial institutions, isn't it? Does the advent出现 of big data affect smaller organizations or individuals个体?

I argue that it does. Take photos, for example. My wife's grandfather was an avid photographer狂热的摄影师 and took photographs throughout his adult life. His entire corpus of mediumformat, slide, and 35mm film,他所有的影集收藏 when scanned in at high resolution高分辨率扫描, occupies around 10 gigabytes. Compare this to the digital photos my family took in 2008, which take up about 5 gigabytes of space. My family is producing photographic data at 35 times the rate my wife's grandfather's did, and the rate is increasing every year as it becomes easier to take more and more photos.

More generally, the digital streams that individuals are producing are growing apace. **Microsoft Research's MyLifeBits project** gives a glimpse一瞥 of the archiving of personal information that may become commonplace in the near future. MyLifeBits was an experiment实验 where an individual's interactions个体的交流—phone calls, emails, documents—were captured electronically and stored for later access. The data gathered聚集 included a photo taken every minute, which resulted in an overall全部的 data volume of

1 gigabyte per month. When storage costs come down enough to make it feasible可行的 to store continuous audio and video, the data volume for a future MyLifeBits service will be many times that.

The trend is for every individual's data footprint足迹 to grow, but perhaps more significantly 显著地, the amount of data generated by machines as a part of the Internet of Things will be even greater than that generated by people机器产生的数据量比人产生的多得多. Machine logs, RFID射频识别 readers, sensor传感器 networks, vehicle车辆 GPS traces痕迹, retail零售 transactions处理—all of these contribute to the growing mountain of data.

The volume of data being made publicly available increases every year, too. Organizations no longer have to merely仅仅 manage their own data; success in the future will be dictated 口述的 to a large extent范围 by their ability to extract提取 value from other organizations' data.

Initiatives举措 such as **Public Data Sets on Amazon Web Services** and **Infochimps.org** exist to foster培养 the "information commons," where data can be freely (or for a modest适度的 price) shared for anyone to download and analyze. Mashups混合 between different information sources make for unexpected意外的 and hitherto迄今 unimaginable applications.

Take, for example, the **Astrometry.net project**, which watches the Astrometry group on Flickr for new photos of the night sky. It analyzes each image and identifies which part of the sky it is from, as well as any interesting celestial bodies天体, such as stars or galaxies 星系. This project shows the kinds of things that are possible when data (in this case, tagged photographic images) is made available and used for something (image analysis) that was not anticipated预料 by the creator.

It has been said that "more data usually beats better algorithms嘻," which is to say that for some problems (such as recommending movies or music based on past preferences), however fiendish极坏的 your algorithms, often they can be beaten simply by having more data (and a less sophisticated复杂的 algorithm).[3]

The good news is that big data is here. The bad news is that we are struggling to store and analyze it.

> [1]. These statistics统计 were reported in a study entitled命名为 **"The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things."** ↩

> [2]. All figures are from 2013 or 2014. For more information, see Tom Groenfeldt, **"At NYSE, The Data Deluge Overwhelms Traditional Databases"**; Rich Miller,

“Facebook Builds Exabyte Data Centers for Cold Stor - age”; Ancestry.com’s *“Company Facts”**; Archive.org’s** *“Petabox”**; and the** *Worldwide LHC Computing Grid project’s welcome page**. ↩

3. The quote引用 is from Anand Rajaraman’s blog post **“More data usually beats better algorithms,”** in which he writes about the Netflix Challenge. Alon Halevy, Peter Norvig, and Fernando Pereira make the same point in **“The Unreasonable**不合理的 **Effectiveness of Data,”** IEEE Intelligent Systems, March/April 2009. ↩

Data Storage and Analysis

The problem is simple: although the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives—have not kept up读数据的速度提升没有跟上硬盘存储能力的增长. One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s,[1] so you could read all the data from a full drive in around five minutes. Over 20 years later, 1-terabyte drives are the norm基准, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk.

This is a long time to read all data on a single drive—and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth百分之一 of the data. Working in parallel平行的, we could read the data in under two minutes.

Using only one hundredth of a disk may seem wasteful. But we can store 100 datasets数据集, each of which is 1 terabyte, and provide shared access to them. We can imagine that the users of such a system would be happy to share access in return for shorter analysis times, and statistically从统计角度, that their analysis jobs would be likely to be spread展开 over time, so they wouldn't interfere干扰 with each other too much.

There's more to being able to read and write data in parallel to or from multiple disks, though.

The first problem to solve is hardware failure: as soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through replication重复，自我复制: redundant冗余的 copies of the data are kept by the system so that in the event of failure, there is another copy available. This is how RAID works, for instance, although Hadoop's filesystem, the Hadoop Distributed Filesystem (HDFS), takes a slightly略微的 different approach, as you shall see later.

The second problem is that most analysis tasks need to be able to combine合并 the data in some way, and data read from one disk may need to be combined with data from any of the other 99 disks. Various distributed分布 systems allow data to be combined from multiple sources, but doing this correctly is notoriously众所周知地 challenging. MapReduce provides a programming model that abstracts抽象 the problem from disk reads and writes, transforming转化 it into a computation计算 over sets of keys and values. We look at the details of this model in later chapters, but the important point for the present discussion is

that there are two parts to the computation—the map and the reduce—and it's the interface between the two where the "mixing" occurs. Like HDFS, MapReduce has built-in reliability可靠性.

In a nutshell简而言之, this is what Hadoop provides: a reliable可靠的, scalable可扩展的 platform for storage and analysis. What's more, because it runs on commodity hardware不贵的硬件 and is open source, Hadoop is affordable.

> [1]. These specifications are for the Seagate ST-41600n. ↩

Querying All Your Data

The approach taken by MapReduce may seem like a brute-force蛮力的 approach. The premise前提 is that the entire dataset—or at least a good portion部分 of it—can be processed for each query. But this is its power. MapReduce is a batch query processor, and the ability to run an ad hoc特别的 query against your whole dataset and get the results in a reasonable time is transformative有改革能力的. It changes the way you think about data and unlocks data that was previously archived on tape or disk. It gives people the opportunity to innovate with data. Questions that took too long to get answered before can now be answered, which in turn leads to new questions and new insights.

For example, Mailtrust, Rackspace's mail division部门, used Hadoop for processing email logs. One ad hoc query they wrote was to find the geographic distribution of their users. In their words:

> This data was so useful that we've scheduled the MapReduce job to run monthly and we will be using this data to help us decide which Rackspace托管服务 data centers to place new mail servers in as we grow.

By bringing several hundred gigabytes of data together and having the tools to analyze it, the Rackspace engineers were able to gain an understanding of the data that they otherwise在其他方面 would never have had, and furthermore, they were able to use what they had learned to improve the service for their customers.

Beyond Batch

For all its strengths, MapReduce is fundamentally根本上 a batch processing批处理 system, and is not suitable for interactive analysis. You can't run a query and get results back in a few seconds or less. Queries typically take minutes or more, so it's best for offline use, where there isn't a human sitting in the processing loop waiting for results.

However, since its original incarnation化身, Hadoop has evolved进化 beyond batch processing. Indeed确实, the term "Hadoop" is sometimes used to refer to a larger ecosystem生态系统 of projects, not just HDFS and MapReduce, that fall under the umbrella of infrastructure基础设施 for distributed computing and large-scale data processing. Many of these are hosted by the **Apache Software Foundation**, which provides support for a community of open source software projects, including the original HTTP Server from which it gets its name.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying根本的 storage. HBase provides both online read/write access of individual个体的 rows and batch operations for reading and writing data in bulk大块, making it a good solution for building applications on.

The real enabler促成者 for new processing models in Hadoop was the introduction of YARN (which stands for Yet Another另一种 Resource Negotiator交涉者) in Hadoop 2. YARN is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster.

In the last few years, there has been a flowering of different processing patterns模式 that work with Hadoop. Here is a sample:

- Interactive SQL
    - By dispensing调剂 with MapReduce and using a distributed query engine that uses dedicated专用的 "always on" daemons守护进程 (like Impala) or container容器 reuse (like Hive on Tez), it's possible to achieve low-latency低延迟 responses for SQL queries on Hadoop while still scaling up to large dataset sizes.
- Iterative迭代 processing
    - Many algorithms—such as those in machine learning—are iterative in nature, so it's much more efficient效率 to hold each intermediate中间的 working set in memory, compared to loading from disk on each iteration. The architecture of MapReduce does not allow this, but it's straightforward简单的 with Spark, for

example, and it enables a highly exploratory探究的 style of working with datasets.

- Stream processing
    - Streaming systems like Storm, Spark Streaming, or Samza make it possible to run real-time实时的, distributed computations on unbounded无限的 streams of data and emit发出 results to Hadoop storage or external systems.
- Search
    - The Solr search platform can run on a Hadoop cluster, indexing documents as they are added to HDFS, and serving search queries from indexes stored in HDFS.

Despite尽管 the emergence出现 of different processing frameworks框架 on Hadoop, MapReduce still has a place for batch processing, and it is useful to understand how it works since it introduces several concepts that apply more generally (like the idea of input formats, or how a dataset is split into pieces).

Comparison with Other Systems

Hadoop isn’t the first distributed system for data storage and analysis, but it has some unique properties that set it apart from other systems that may seem similar. Here we look at some of them.

Relational Database Management Systems

Why can't we use databases with lots of disks to do large-scale analysis? Why is Hadoop needed?

The answer to these questions comes from another trend in disk drives: seek time is improving more slowly than transfer rate. Seeking is the process of moving the disk's head to a particular place on the disk to read or write data. It characterizes the latency of a disk operation, whereas the transfer rate corresponds to a disk's bandwidth.

If the data access pattern is dominated by seeks, it will take longer to read or write large portions of the dataset than streaming through it, which operates at the transfer rate. On the other hand, for updating a small proportion of records in a database, a traditional B-Tree (the data structure used in relational databases, which is limited by the rate at which it can perform seeks) works well. For updating the majority of a database, a B- Tree is less efficient than MapReduce, which uses Sort/Merge to rebuild the database.

In many ways, MapReduce can be seen as a complement to a Relational Database Management System (RDBMS). (The differences between the two systems are shown in Table 1-1.) MapReduce is a good fit for problems that need to analyze the whole dataset in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once and read many times, whereas a relational database is good for datasets that are continually updated.[1]

Table 1-1. RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Transactions	ACID	None
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

However, the differences between relational databases and Hadoop systems are blurring. Relational databases have started incorporating some of the ideas from Hadoop, and from the other direction, Hadoop systems such as Hive are becoming more interactive (by moving away from MapReduce) and adding features like indexes and transactions that make them look more and more like traditional RDBMSs.

Another difference between Hadoop and an RDBMS is the amount of structure in the datasets on which they operate. Structured data is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Semi-structured data, on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. Unstructured data does not have any particular internal structure: for xample, plain text or image data. Hadoop works well on unstructured or semi-structured data because it is designed to interpret the data at processing time (so called schema-on-read). This provides flexibility and avoids the costly data loading phase of an RDBMS, since in Hadoop it is just a file copy.

Relational data is often normalized to retain its integrity and remove redundancy. Normalization poses problems for Hadoop processing because it makes reading a record a nonlocal operation, and one of the central assumptions that Hadoop makes is that it is possible to perform (high-speed) streaming reads and writes.

A web server log is a good example of a set of records that is not normalized (for example, the client hostnames are specified in full each time, even though the same client may appear many times), and this is one reason that logfiles of all kinds are particularly well suited to analysis with Hadoop. Note that Hadoop can perform joins; it's just that they are not used as much as in the relational world.

MapReduce—and the other processing models in Hadoop—scales linearly with the size of the data. Data is partitioned, and the functional primitives (like map and reduce) can work in parallel on separate partitions. This means that if you double the size of the input data, a job will run twice as slowly. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

> 1. In January 2007, David J. DeWitt and Michael Stonebraker caused a stir by publishing "**MapReduce: A major step backwards**," in which they criticized MapReduce for being a poor substitute for relational databases. Many commentators argued that it was a false comparison (see, for example, Mark C. Chu-Carroll's "**Databases are hammers; MapReduce is a screwdriver**"), and DeWitt and

> Stonebraker followed up with “MapReduce II,” where they addressed the main topics brought up by others. ↩

Grid Computing

The high-performance computing (HPC) and grid computing communities have been doing large-scale data processing for years, using such application program interfaces (APIs) as the Message Passing Interface (MPI). Broadly, the approach in HPC is to distribute the work across a cluster of machines, which access a shared filesystem, hosted by a storage area network (SAN). This works well for predominantly compute-intensive jobs, but it becomes a problem when nodes need to access larger data volumes (hundreds of gigabytes, the point at which Hadoop really starts to shine), since the network bandwidth is the bottleneck and compute nodes become idle.

Hadoop tries to co-locate the data with the compute nodes, so data access is fast because it is local.[1] This feature, known as data locality, is at the heart of data processing in Hadoop and is the reason for its good performance. Recognizing that network bandwidth is the most precious resource in a data center environment (it is easy to saturate network links by copying data around), Hadoop goes to great lengths to conserve it by explicitly modeling network topology. Notice that this arrangement does not preclude high-CPU analyses in Hadoop.

MPI gives great control to programmers, but it requires that they explicitly handle the mechanics of the data flow, exposed via low-level C routines and constructs such as sockets, as well as the higher-level algorithms for the analyses. Processing in Hadoop operates only at the higher level: the programmer thinks in terms of the data model (such as key-value pairs for MapReduce), while the data flow remains implicit.

Coordinating the processes in a large-scale distributed computation is a challenge. The hardest aspect is gracefully handling partial failure—when you don’t know whether or not a remote process has failed—and still making progress with the overall computation. Distributed processing frameworks like MapReduce spare the programmer from having to think about failure, since the implementation detects failed tasks and reschedules replacements on machines that are healthy. MapReduce is able to do this because it is a shared-nothing architecture, meaning that tasks have no dependence on one other. (This is a slight oversimplification, since the output from mappers is fed to the reducers, but this is under the control of the MapReduce system; in this case, it needs to take more care rerunning a failed reducer than rerunning a failed map, because it has to make sure it can retrieve the necessary map outputs and, if not, regenerate them by running the relevant maps again.) So from the programmer’s point of view, the order in which the tasks run

doesn’t matter. By contrast, MPI programs have to explicitly manage their own checkpointing and recovery, which gives more control to the programmer but makes them more difficult to write.

> [1]. Jim Gray was an early advocate of putting the computation near the data. See “**Distributed Computing Economics**,” March 2003. ↩

Volunteer Computing

When people first hear about Hadoop and MapReduce they often ask, “How is it different from SETI@home?” SETI, the Search for Extra-Terrestrial Intelligence, runs a project called SETI@home in which volunteers donate CPU time from their otherwise idle computers to analyze radio telescope data for signs of intelligent life outside Earth. **SETI@home** is the most well known of many volunteer computing projects; others include the Great Internet Mersenne Prime Search (to search for large prime numbers) and Folding@home (to understand protein folding and how it relates to disease).

Volunteer computing projects work by breaking the problems they are trying to solve into chunks called work units, which are sent to computers around the world to be analyzed. For example, a SETI@home work unit is about 0.35 MB of radio telescope data, and takes hours or days to analyze on a typical home computer. When the analysis is completed, the results are sent back to the server, and the client gets another work unit. As a precaution to combat cheating, each work unit is sent to three different machines and needs at least two results to agree to be accepted.

Although SETI@home may be superficially similar to MapReduce (breaking a problem into independent pieces to be worked on in parallel), there are some significant differences. The SETI@home problem is very CPU-intensive, which makes it suitable for running on hundreds of thousands of computers across the world[1] because the time to transfer the work unit is dwarfed by the time to run the computation on it. Volunteers are donating CPU cycles, not bandwidth.

MapReduce is designed to run jobs that last minutes or hours on trusted, dedicated hardware running in a single data center with very high aggregate bandwidth interconnects. By contrast, SETI@home runs a perpetual computation on untrusted machines on the Internet with highly variable connection speeds and no data locality.

> [1]. In January 2008, **SETI@home was reported** to be processing 300 gigabytes a day, using 320,000 computers (most of which are not dedicated to SETI@home; they are used for other things, too). ↩

A Brief History of Apache Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

The Origin of the Name “Hadoop”

The name Hadoop is not an acronym; it’s a made-up name. The project’s creator, Doug Cutting, explains how the name came about:

> The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid’s term.

Projects in the Hadoop ecosystem also tend to have names that are unrelated to their function, often with an elephant or other animal theme (“Pig,” for example). Smaller components are given more descriptive (and therefore more mundane) names. This is a good principle, as it means you can generally work out what something does from its name. For example, the namenode[1] manages the filesystem namespace.

Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It’s expensive, too: Mike Cafarella and Doug Cutting estimated a system supporting a one-billion-page index would cost around $500,000 in hardware, with a monthly running cost of $30,000.[2] Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, its creators realized that their architecture wouldn’t scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google’s distributed filesystem, called GFS, which was being used in (P12)

[1] In this book, we use the lowercase form, “namenode,” to denote the entity when it’s being referred to generally, and the CamelCase form NameNode to denote the Java class that implements it. [2] See Mike Cafarella and Doug Cutting, “**Building Nutch: Open Source Search**,” ACM Queue, April 2004.

What's in This Book

Chapter 2. MapReduce

A Weather Dataset

Data Format

Analyzing the Data with Unix Tools

Analyzing the Data with Hadoop

Map and Reduce

Java MapRedece

Scaling Out

Data Flow

Combiner Functions

Running a Distributed MapReduce Job

Hadoop Streaming

Ruby

Python

Chapter 3. The Hadoop Distributed Filesystem

THe Design of HDFS

HDFS Concepts

Blocks

Namenodes and Datanodes

Block Caching

HDFS Federation

HDFS High Availability

The Command-Line Interface

Basic FileSystem Operations

Hadoop Filesystems

Interfaces

The Java Interface

Reading Data from a Hadoop URL

Reading Data Using the FileSystem API

Writing Data

Directories

Querying the Filesystem

Deleting Data

Data Flow

Anatomy of a File Read

Anatomy of a File Writw

Coherency Model

Parallel Copying with distcp

Keeping an HDFS Cluster Balanced

Chapter 4. YARN

Anatomy of a YARN Application Run

Resource Requests

Application Lifespan

Building YARN Application

Scheduling in YARN

Scheduler Options

Capacity Scheduler Configuration

Fair Schedular Configuration

Delay Scheduling

Dominant Resource Fairness

Futher Reading

Part II. MapReduce

Chapter 6. Developing a MapReduce Application

The Configuration API

Combining Resources

variable Expansion

Setting Up the Development Environment

Managing Configuration

GenericOptionsParser, Tool, and ToolRunner

Writing a Unit Test with MRUnit

Mapper

Reducer

Running Locally on Test Data

Running a Job in a Local Job Runner

Testing the Driver

Running on a Cluster

Packing a Job

Luanching a Job

The MapReduce Web UI

retrieving the Results

Debugging a Job

hadoop Logs

Remote Debugging

Tuning a Job

Profiling Tasks

MapReduce Workflows

Decomposing a Problem into MapReduce Jobs

JobControl

Apache Oozie

Part IV. Related Projects

Chapter 19. Spark

Installing Spark

An Example

Spark Application, Jobs, Stages, and Tasks

A Scala Standalong Application

A Java Example

A Python Example

Resilient Distributed Datasets

Creation

Transformations and Actions

Persistence

Serialization

Shared Variables

Broadcast Variables

Accumulators

Anatomy of a Spark Job Run

Job Submission

DAG Construction

Task Scheduling

Task Execution

Execution and Cluster Managers

Spark on YARN

Futher Reading